September 1992

Report No. STAN-CS-92-1449
CSL-TR-92-547
Thesis

Accurate Analysis of Array References

by

Dror Eliezer Maydan

Department of Computer Science

Stanford University
Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	9/22/92	TR

4. TITLE AND SUBTITLE

ACCURATE ANALYSIS OF ARRAY REFERENCES

5. FUNDING NUMBERS

N00039-91-C0138

6. AUTHOR(S)

DROR ELIEZER MAYDAN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

STANFORD UNIVERSITY
CSL, CIS, Stanford, CA 94305-4070

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA/CSTO, 3700 N. Fairfax, Arlington, VA 22203-1714

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

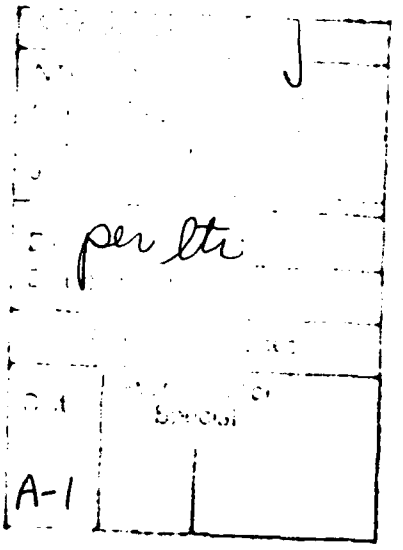
This thesis addresses the problem of data dependence analysis, the base step in detecting loop level parallelism in scientific programs.
Traditional data dependence analysis research has concentrated on the simpler problem of affine memory disambiguation. Many algorithms have been developed that conservatively approximate even this simpler problem. Using a series of algorithms, each one guaranteed to be exact for a certain class of input, we are able to devise a new method that in practice solves exactly and efficiently the affine memory disambiguation problem. Because our system is exact, we can devise an experiment to test the effectiveness of affine memory disambiguation at approximating the full dependence problem. We discover that the lack of data-flow information on array elements is the key limitation of affine memory disambiguators. We develop a new representation and algorithm to efficiently calculate these data-flow dependences. Finally, we address the problem of interprocedural data dependence analysis. By using an array summary representation that is guaranteed to be exact when applicable, we can combine summaries with inlining to exactly and efficiently analyze affine array references across procedure boundaries. Taken together, our algorithms generate the more accurate information that will be needed to exploit parallelism in the future.

14. SUBJECT TERMS

15. NUMBER OF PAGES 144

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT



ACCURATE ANALYSIS OF ARRAY REFERENCES

Dror Eliezer Maydan

Technical Report: CSL-TR-92-547

September 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Modern computer systems are increasingly relying on parallelism to improve performance. Automatic parallelization techniques offer the hope that users can simply and portably exploit parallelism. This thesis addresses the problem of data dependence analysis, the base step in detecting loop level parallelism in scientific programs. Exploiting parallelism can change the order of memory operations. Data dependence analysis involves analyzing the dynamic memory reference behavior of array operations so that compilers will only parallelize loops in the cases where any resultant reordering of memory references does not change the sequential semantics of the program.

In general, data dependence analysis is undecidable, and compilers must conservatively approximate array reference behavior, thus sequentializing parallel loops. Traditional data dependence analysis research has concentrated on the simpler problem of affine memory disambiguation. Many algorithms have been developed that conservatively approximate even this simpler problem. By using a series of algorithms, each one guaranteed to be exact for a certain class of input, we are able to devise a new method that in practice solves exactly and efficiently the affine memory disambiguation problem. Because our affine memory disambiguator is exact in practice, we can devise an experiment to test the effectiveness of affine memory disambiguation at approximating the full data dependence problem. We discover that the lack of data-flow information on array elements is the key limitation of affine memory disambiguators. We develop a new representation and algorithm to efficiently calculate these data-flow dependences. Finally, we address the problem of interprocedural data dependence analysis. By using an array summary representation that is guaranteed to be exact when applicable, we can combine summary information with inlining to exactly and efficiently analyze affine array references across procedure boundaries. Taken together, our algorithms generate the more accurate information that will be needed to exploit parallelism in the future.

Acknowledgements

I would like to thank all the people who have made writing this thesis a more pleasant task. In particular, I'd like to thank my principal advisor, John Hennessy, who introduced me to this topic and gave me broad support and guidance throughout my time at Stanford. I'd like to thank Monica Lam for devoting a countless amount of energy and support to many parts of this thesis and Anoop Gupta for his valuable comments given while serving on my reading committee.

There are many other people whom I would like to thank. James Larus gave me invaluable help with the llpp system used in Chapter 3. Chapter 4 is joint work with Saman Amarasinghe who is a pleasure to work with. I feel privileged to have been a part of an excellent research team and would like to acknowledge some of its other members: Saman Amarasinghe, Jennifer Anderson, Rob French, Aaron Goldberg, Amy Lim, Karen Pieper, Martin Rinard, Ed Rothberg, Kevin Rudd, Dan Scales, Mike Smith, Steve Tjiang, Bob Wilson and Michael Wolf.

Last, but certainly not least, I'd like to thank my family and all my friends whose unconditional support made this thesis possible.

I would also like to acknowledge support from an AT&T Bell Laboratories fellowship and from DARPA grants N00014-87-K-0828 and N00039-91-C-0138.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Powerful computer systems are increasingly using parallelism to provide increased performance. Today we are witnessing the introduction of massively parallel machines with multi-level parallelism. Machines such as the Intel Touchstone and Thinking Machines' CM-5 consist of large numbers of microprocessors, each processor itself having some limited amount of parallelism.

Utilizing these machines effectively is becoming increasingly complex. As a result, much work has gone into developing parallelizing compilers for scientific programs. These compilers attempt to detect the parallelism present in the original source and transform the program to effectively utilize the machine's resources. The key to finding parallelism in scientific programs is analyzing array references, which is called data dependence analysis.

1.1 Data Dependence Analysis

Most of the parallelism found in scientific programs is present in loops. One common method to exploit this parallelism is to execute the loop iterations concurrently.

In this first section, we give an overview describing the type of information that is required to parallelize loops. We base our framework on material found in [53] and [55]. Their descriptions are based on a long line of work originating from Kuck, Lamport and their associates. In the next section, we discuss calculating the information necessary for

parallelization.

Before parallelizing a loop, we must be certain that the parallelized version will retain the sequential semantics of the original version. Current compiler systems conservatively guarantee this condition by requiring that parallelization maintain the serial execution order between every write operation and every other write or read operation to the same memory location [55].

Parallelizing loops under this model requires the compiler to analyze array reference patterns. Consider the following example.

```
do i = 11 to 20
  a[i] = a[i-1]+3
end do
do i = 11 to 20
  a[i] = a[i-10]+3
end do
```

We would like to be able to run all the iterations of each loop in parallel, but in the first loop the location being read in each iteration was written in the previous iteration. It is not possible to execute this loop in parallel and still guarantee the sequential semantics.

In the second loop, the locations being written do not overlap the locations being read. Each iteration, is reading from and writing to different locations. The values being read were written before the first iteration of the loop. It is therefore possible to execute the iterations concurrently.

Two array references a and a′ are said to be *dependent* if any of the locations accessed by reference a are also accessed by reference a′ [53]. Otherwise, the two references are *independent*. The existence of a dependence implies that there exist two iterations of the loop, $\vec{i}$ and $\vec{i'}$, within the loop bounds, such that the location accessed by reference a in iteration $\vec{i}$ is the same as the location accessed by reference a′ in iteration $\vec{i'}$. We introduce the term *dependent iteration pairs* to describe the set of such pairs, $(\vec{i},\vec{i'})$.

If all array reference pairs in a loop are independent, we can run all the iterations of the loop in parallel. If some pairs are dependent, it might still be possible, depending on

the nature of the dependences, to run the iterations in parallel. For example:

```
do i = 11 to 20
  a[i] = a[i]+3
end do
```

These references are dependent since the same locations are both written and read. For each dependent iteration pair, though, we know that $\vec{i} = \vec{i'}$. There are no dependences between operations in different iterations, and although we cannot run the read and the write operations of the same iteration concurrently, we can correctly run all the separate iterations in parallel.

Any dependent iteration pair such that $\vec{i} = \vec{i'}$ is a *loop-independent dependence*, while a dependent iteration pair such that $\vec{i} \neq \vec{i'}$ is a *loop-carried dependence* [55]. All the iterations of a loop nesting can be run in parallel if and only if there are no loop-carried dependences between any two references in the loop.

To parallelize a subset of the loop nestings, we need to know more than if there are loop-carried dependences. For example:

```
do i = 11 to 20
  do j = 11 to 20
    a[i][j] = a[i-1][j-1]+3
  end do
end do
```

All the dependences in this loop nesting are loop-carried. Therefore we cannot run all the iterations in parallel. Nonetheless, if we run loop i sequentially, we can run the j loop in parallel. For a given value of i, the array operations in all the iterations of the j loop refer to different locations.

To fully exploit all the parallelism inherently present in a loop, we need to calculate all the dependent iteration pairs. This is both infeasible and unnecessary for most optimization techniques. *Distance vectors* and *direction vectors* are standard representations

that allow us to summarize the set of dependent iteration pairs [55]. Distance vectors represent the vector difference between the two iteration elements in a dependent iteration pair. Two references are dependent with distance vector $\vec{d}$ if there exists a dependent iteration pair, $(\vec{i}, \vec{i'})$, such that $\vec{i'} - \vec{i} = \vec{d}$. In the above example, there is a dependence from the write to the read with distance (1,1). Direction vectors represent the sign of the distance vectors. We replace each component distance with its sign; $+$, $-$ or 0. We use a $*$ as a short hand notation to represent the direction when the sign of the distance is unknown or when there are dependent iteration pairs with all the possible directions. In the above example, there is a dependence from the write to the read with direction (+,+). It can be shown that since there is no dependence with direction (0,+), it is possible to run loop j in parallel as long as we sequentialize loop i. We use the term *dependence vector* to refer to either distance or direction vectors.

Standard compiler systems require parallelization to preserve the order between all write operations and all read/write operations to the same location [55]. Given this model, distance and direction vectors are sufficient representations for parallelization. This model, though, is too restrictive. Not all dependences are equally harmful. The only dependences that inherently limit parallelism are dependences between a write operation and a read operation where the read uses a value produced by an earlier write. All other dependences are artifacts of aliasing. Take, for example, the following two loops.

```
Loop1:
do i = 11 to 20
  a[i] = a[i-1]+3
end do

Loop2:
do i = 11 to 20
  a[i] = a[i+1]+3
end do
```

There are loop-carried dependences in both loops, and we cannot run either loop,

without modifications, in parallel. In the first loop, the value being written in iteration i is being consumed in the next iteration $i + 1$. The loop is inherently sequential. In the second loop, the location being read in iteration i is being overwritten in the next iteration $i + 1$. There is no data being transferred across iterations of the loop. While we cannot run the loop as written in parallel, we can transform it into the following two loops, each of which is individually parallelizable.

```
do i = 12 to 21
  b[i] = a[i]
end do
do i = 11 to 20
  a[i] = b[i+1]+3
end do
```

The use of distance vectors is sufficient to discover that Loop1 is inherently sequential while Loop2 is not. In Loop1, there is a dependence with a distance vector of (1) from the write statement to the read statement while in Loop2 there is a dependence with a distance vector of (-1). For any pair of write and read references with a positive distance vector from the write to the read, i.e. $\vec{i}_r - \vec{i}_w > 0$, the write operation occurs dynamically before the read, and the dependence is called a *true dependence*. Any pair for which the distance vector is negative is called an *anti-dependence* [26][27]. Anti-dependences are artifacts of aliasing and can always be eliminated. Note that if the distance vector is 0, the dependence is a true dependence if the write comes lexically before the read and an anti-dependence otherwise.

While only true dependences inherently limit parallelism, not all true dependences are inherently harmful. For example:

```
do i = 11 to 20
  do j = 11 to 20
    a[j] = ...
  do j = 11 to 20
    ... = a[j]
```

There is a loop-carried true dependence between the write and the read because the locations written in iteration $i = x$ are read in all the later iterations $i = x+1, x+2, \ldots, 20$. Note, though, that in each iteration of the i loop, the same locations are being overwritten. Thus the value read in iteration $i = x$ was in fact written in the same iteration. The array a is in fact being used as a temporary array. We define a read reference to be *data-flow dependent* on a write reference if the write reference writes a *value* that is read by an instance of the read reference. Data-flow dependences are a subset of true dependences. Only data-flow dependences inherently limit parallelism. In the above example, there are no loop-carried data-flow dependences, and we can parallelize the loop by *privatizing* the array, giving each processor its own private copy of the temporary array.

Dependence vectors do not contain sufficient information to distinguish true dependences from data-flow dependences. A new representation, *data-flow dependence vectors*, is required. Data-flow dependence vectors will be described in detail in Chapter 4.

1.2 Calculating Data Dependence Relations

Ideally one would always like to know the exact dependence and data-flow dependence vectors between any two references. Unfortunately, even simpler problems, such as deciding if two references are dependent, are undecidable at compile time in the general case. For example:

```
read (n)
do i = 11 to 20
  a[i] = a[i-n]+3
end do
```

Whether or not the two references are dependent depends on the value of n, which is unknown at compile time.

Even ignoring the issue of static versus dynamic dependences, the problem can be made arbitrarily difficult. For example:

```
if n > 2 then
  if a > 0 then
    if b > 0 then
      if c > 0 then
        x[a^n] = x[b^n + c^n]+3
      end if
    end if
  end if
end if
```

To prove that these two references are always independent, a compiler would have to prove Fermat's last theorem.

1.2.1 Affine Memory Disambiguation

Traditional approaches restrict the data dependence domain to the simpler problem of affine memory disambiguation [8][53][55]. Pairs of references that cannot be proved independent in this domain are assumed dependent. Before discussing more accurate approaches, we shall first describe the affine memory disambiguation model.

A memory disambiguator does not distinguish true dependences from data-flow dependences. A memory disambiguator calculates whether two references refer to overlapping locations; it does not calculate the flow of values through a program.

An affine data dependence solver only utilizes information from loop bounds that are integer linear functions of more outwardly nested loop indices and from array reference functions that are integer linear functions of the loop indices. This is frequently extended to allow constant symbolic terms as well.

Affine systems allow more general loops and array references than we have shown in the previous examples. They can handle multi-dimensional arrays and nested loops. The loops need not be rectangular; they can be triangular or trapezoidal. For example:

```
do i = 10 to 20
  do j = 10 to i+3
    a[i][j] = a[3*j][2*i-1]+3
  end do
end do
```

An affine system is not able to handle other cases such as nonlinear terms or indirect array references. The following example is not affine.

```
do i = 10 to n
  a[i^2] = a[3]+3
end do
```

The first array reference function, i^2, is not linear in i. Note that the two references are independent since there is no integer i such that $i^2 = 3$, but an affine system will have to assume that these two references are truly dependent.

Most work in data dependence analysis has focused on solving the affine memory disambiguation problem. Most previous approaches to even this simplified problem have approximated its solution. In Chapter 2, we develop a system that exactly and efficiently solves the affine memory disambiguation problem in practice. Solving this problem exactly allows us to judge how effectively affine memory disambiguation approximates data dependence analysis. In Chapter 3, we develop a method for judging the effectiveness of this approximation. We use the llpp system developed by Larus [28], a dynamic trace-based system, to find all the intraprocedural data dependences dynamically. By comparing the results of our system to this dynamic system, we show that while the affine approximation is reasonable, memory disambiguation is not.

Feautrier has developed an algorithm that can be used to differentiate data-flow dependences from other true dependences in the domain of simple loop nests that contain no IF statements, no subroutine calls and no non-affine terms [16][15]. His algorithm is too expensive to be used in real compiler systems. In Chapter 4, we develop a new algorithm

for Feautrier's domain that is much more efficient and is as accurate as Feautrier's in the vast majority of cases seen in practice.

1.2.2 Interprocedural Analysis

To fully utilize all the possible parallelism, one would have to compare every array write reference to every other array reference. While this is frequently done for a single subroutine, interprocedural analysis greatly increases the number of comparisons required. To be completely accurate, one must inline every subroutine call to make the comparisons. As we will show in Chapter 5, this is impractical. Most systems either partially inline a program or attempt to summarize a set of accesses. Both approaches are inexact and potentially sacrifice some parallelism. Partially inlining a program prevents parallelization across remaining subroutine calls. Summarizing a set of accesses can limit parallelism when the summary is not accurate. In addition, if a summary prevents the parallelization of a loop, it is not possible to know if the summary is conservatively faulty or if the loop is inherently sequential. It is therefore very difficult to judge the effectiveness of a particular summary algorithm.

In Chapter 5, we develop an algorithm for interprocedural parallelization that combines summary information with inlining. Whenever our summary cannot describe the array accesses exactly, we inline. Thus we are able to use a simple summary structure that can be efficiently computed while retaining the accuracy of full inlining.

Finally, in Chapter 6, we summarize our work and discuss future directions.

Chapter 2

Affine Memory Disambiguation

We showed in Chapter 1 that it is not possible to solve the data dependence analysis problem exactly at compile time. There are pairs of independent references for which the compiler must assume dependences exist. Traditional approaches restrict the data dependence domain to the simpler, decidable, problem of affine memory disambiguation [8][53]. Pairs of references that cannot be proved independent in this domain are assumed dependent.

In this chapter, we present an approach to solving the affine memory disambiguation problem [37]. While in the worst case affine memory disambiguation is too expensive to solve exactly, we show that in the cases seen in practice we can be both exact and efficient. Even when computing distance and direction vectors and when allowing symbolic constants, we are able to be exact in *every* case we have seen in practice at a very reasonable cost.

2.1 Problem Definition

We first give the standard definition for affine memory disambiguation. The affine domain restricts us to only utilize constraints that come from loop bounds that are integral linear functions of more outwardly nested loop indices and array references that are integral linear functions of the loop indices.

For the purpose of this thesis, we use a more generalized form of the affine restriction,

where the affine conditions do not necessarily have to be met directly in the source program. For example, we can use optimization techniques (constant propagation and induction variable and forward substitution [49][50]) to increase the applicability of the solution techniques. For example:

$n = 100$
$i_2 = 0$
...
do i = 1 **to** 10
  $i_2 = i_2 + 2$
  a[$i_2 + n$] = a[$i_2 + 2 * n + 1$]+3
end do

These references are not strictly affine since they refer to the non-loop indices i_2 and n. Nonetheless, our optimizer will discover that $n = 100$ and that $i_2 = 2i$, and it will transform the code into:

do i = 1 **to** 10
  a[$2 * i + 100$] = a[$2 * i + 201$]+3
end do

which does meet our conditions for analysis.

In addition, having non-affine terms that we cannot eliminate implies that we lose information, but it does not necessarily force us to assume dependence. As long as some of the array dimensions are affine, we may be able to use these dimensions to prove independence. For example:

do i = 10 **to** 20
  a[i^2] [$2 * i$] = a[3] [$2 * i + 1$]+3
end do

The first dimension of the array write contains a non-affine term, but the second dimension of both references are affine. Looking only at the second dimension of the

array, we know that the write only refers to even locations while the read only refers to odd ones. Even without using the information in the first dimension, we are able to prove independence.

In the memory disambiguation domain, two references are dependent if and only if the sets of locations they refer to overlap. That is, two references are dependent if there exists a pair of iterations (i,i') such that the location accessed by one of the references in iteration i is the same location accessed by the other reference in iteration i'.

We can formally define dependence in the affine memory disambiguation domain as follows:

Definition 2.1.1 *Given the following general normalized loop (we normalize the step size to 1):*

do $i_1 = L_1$ **to** U_1
  do $i_2 = L_2(i_1)$ **to** $U_2(i_1)$
    ...
    do $i_n = L_n(i_1, \ldots, i_{n-1})$ **to** $U_n(i_1, \ldots, i_{n-1})$
      a$[f_1(\vec{i})]\,[f_2(\vec{i})]\ldots[f_m(\vec{i})] = \ldots$
      $\ldots =$a$[f'_1(\vec{i})]\,[f'_2(\vec{i})]\ldots[f'_m(\vec{i})]$
    end do
  end do
end do

such that all the L, U, f, f' are known integer, linear functions. The two references are **dependent** *in the affine memory disambiguation domain iff*

$\exists$ integer $i_1,\ldots,i_n,i'_1,\ldots,i'_n$ such that
$f_1(\vec{i}) = f'_1(\vec{i'}),\ \ldots,\ f_m(\vec{i}) = f'_m(\vec{i'})$
$L_1 \le i_1 \le U_1$
$L_1 \le i'_1 \le U_1$
...
$L_n(i_1,\ldots,i_{n-1}) \le i_n \le U_n(i_1,\ldots,i_{n-1})$
$L_n(i'_1,\ldots,i'_{n-1}) \le i'_n \le U_n(i'_1,\ldots,i'_{n-1})$

In matrix form this is equivalent to

$$\exists \text{ integral } \vec{x} \text{ such that } A_1\vec{x} = \vec{b_1}, A_2\vec{x} \leq \vec{b_2} \quad (1)$$

where $\vec{x}$ is the combined vector $(\vec{i}, \vec{i'})$, where the rows of A_1 and b_1 are the array reference constraints and where the rows of A_2 and b_2 are the loop bounds constraints. By replacing any equality $ax = b$ in (1) by the two inequalities $ax \leq b$ and $-ax \leq -b$ we see that this is equivalent to

$$\exists \text{ integral } \vec{x} \text{ such that } A\vec{x} \leq \vec{b}. \quad (2)$$

2.1.1 Affine Memory Disambiguation is Integer Programming

Ideally one would like an exact affine memory disambiguation system. Unfortunately, affine memory disambiguation in general is exactly equivalent to integer programming, a well-studied problem. The standard integer programming problem [44] is to find

$$\max \vec{x} \text{ such that } A\vec{x} < \vec{b}, \vec{x} \text{ integral} \quad (3)$$

It is clear that (2) is a special case of (3) so affine memory disambiguation can be reduced to integer programming. Another polynomially equivalent version of integer programming is [44]

$$\exists \vec{x} \text{ such that } A\vec{x} = \vec{b}, \vec{x} \geq 0, \vec{x} \text{ integral} \quad (4)$$

If A is an $m \times n$ matrix, one can reduce (4) to affine memory disambiguation by constructing the following program:

do x_1 = 0 **to** *unknown*

...

do x_n = 0 **to** *unknown*

a[$A_{1,1}x_1 + \ldots A_{1,n}x_n$]...[$A_{m,1}x_1 + \ldots$] = ...

... = a[b_1]...[b_m]

Given that we showed that (2) can be reduced to (3) and that (4) can be reduced to (2), the affine memory disambiguation and the integer programming problems are polynomially equivalent. All integer programming algorithms that we know of are too expensive in the worst case. Integer programming is NP-Complete; any existing algorithm either depends exponentially on the number of variables and constraints or depends linearly on the size of the coefficients. Typical programs have very few array dimensions and do not have deeply nested loops. Therefore being exponential in the number of variables and constraints could be acceptable. The size of the coefficients, though, could conceivably be very large.

The complexity of most common algorithms (branch and bound, cutting plane) depend on the size of the coefficients in the worst case. Lenstra [29] and Kannan [24] have developed algorithms that do not depend on the coefficients, but in the worst case, Kannan's algorithm is $O(n^{O(n)})$ where n is the number of variables. Unfortunately, even for the relatively small n we see in practice, this is much too expensive to use in compilers. Therefore, we do not believe it is possible to develop a practical test that will apply to every conceivable case. Nonetheless, as we next show, special case algorithms that are efficient, apply to all the examples we have found in practice.

2.2 Solving the Affine Memory Disambiguation Problem

Many algorithms have been proposed for affine memory disambiguation, each one selecting different tradeoffs between accuracy and efficiency. Traditional algorithms such as the GCD Test and Banerjee's inequalities attempt to prove independence, assuming dependence if they fail [4][8][53]. We call these algorithms *may* algorithms since when they return dependent, we do not know if an approximation was made.

In its most general form, the GCD Test solves the affine memory disambiguation problem exactly ignoring the loop bounds. For example, the GCD Test could conclude that the two references $2i$ and $2i + 1$ are independent since they are in fact independent regardless of the values of the loop bounds. If there are two references that are only independent because of the values of the bounds, then the GCD Test must incorrectly assume that the two references are dependent. Often, a simpler and less accurate version

of the GCD Test is used. We will describe both versions in greater detail in the next section.

Banerjee's inequalities use the loop bounds to find a range of values for each dimension of each reference. If the ranges of the two references overlap in each dimension, they are assumed dependent. Otherwise, they are known to be independent. If every loop bound is a constant and the different array dimensions are not coupled (no loop index is used in more than one dimension), Banerjee's inequalities give the exact solution to the real problem

$$\exists\ \vec{x} \text{ such that } A\vec{x} \leq \vec{b}$$

If the set of equations has a real solution, but no integer one, Banerjee's inequalities must incorrectly assume that the references are dependent. When some of the loop bounds are triangular or trapezoidal or when multiple dimensions are coupled such as `a[i][i]`, Banerjee further approximates and does not solve the real (non-integer) problem exactly either.

Other researchers have developed more accurate *may* algorithms. Triolet uses the Fourier-Motzkin algorithm [51]. Fourier-Motzkin solves the real system exactly, even when there are triangular and trapezoidal bounds. Wallace has developed a variation of simplex extended to look for integer solutions [52]. To guarantee termination, it must assume dependence after a certain number of iterations. The Lambda Test extends Banerjee's inequalities to better deal with coupled subscripts [32]. The Power Test [54] combines the GCD Test with Fourier-Motzkin.

While some of these algorithms may be accurate in most cases, they all have the problem that their approximations are implicit. If such an algorithm returns dependent, we do not know if an approximation was made. In addition, there is evidence that some of these algorithms are too inefficient. Li, Yew and Zhu, for example, consider Fourier-Motzkin to be too expensive [32].

Some work has been done on algorithms that are guaranteed to be exact for special case inputs. Simple loop residue [46], Li and Yew's work [31], the I Test [25] and the

Delta Test [18] fall into this category. Little work has been done to analyze either the accuracy or the efficiency of these algorithms in practice.

None of these algorithms, as far as we know, has been definitively shown to be both "accurate enough" and "efficient enough". In fact, Shen, Li and Yew found that cases such as coupled subscripts appear frequently and cannot be analyzed accurately using traditional algorithms such as Banerjee's [45].

Our approach is to use a series of special case exact tests. If the input is not of the appropriate form for an algorithm, then we try the next one. Using a series of tests allows us to be exact for a wider range of inputs. We evaluated our algorithms on the PERFECT Club [14], a set of 13 scientific benchmarks, and found the algorithms to be exact in every case.

Cascading exact tests can also be much more efficient than cascading inexact ones. By attempting our most applicable and least expensive test first, in most cases, even the dependent ones, we can return a definitive answer using just one exact test. Even when one test is not sufficient, we only need to check the applicability of multiple tests. We never have to apply more than one. In contrast, cascading *may* algorithms would require using all the tests on at least all the dependent cases. As we will show, most cases encountered in practice are in fact dependent. In fact, most direction vectors tested are dependent as well.

2.3 Data Dependence Tests

In this section, we describe the individual tests used in our approach: the Extended GCD Test, the Single Variable Per Constraint Test, the Acyclic Test, the Simple Loop Residue Test and Fourier-Motzkin Elimination. We describe them in the order in which they are applied by our system.

2.3.1 Extended GCD Test

We use the Extended GCD Test [8] as a preprocessing step for our other tests. While the test itself is not exact, it allows us to transform our problem into a simpler and smaller

form, increasing the applicability of our other tests. This test solves the simpler question: Ignoring the bounds, is there an integral solution to the set of equations. From equation (1) we see that this is equivalent to: does there exist an integer vector $\vec{x}$ such that $A\vec{x} = b$. If this system of equations is inconsistent, then we know that the original system is also inconsistent since the loop bounds merely introduce additional constraints. If the system is consistent, then the total system may be either inconsistent or consistent. In this case, though, we are able to use the results of the extended GCD Test to make a change of variables that simplifies the original problem. The original GCD Test is derived from number theory. The single equation $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$ has an integer solution iff $\gcd(a_i)$ divides b. This gives us an exact test for single-dimensional arrays ignoring bounds. Banerjee shows how this can be extended to multi-dimensional arrays. The system of equations $\vec{x}A = \vec{c}$ can always be factored into a unimodular[1] integer matrix U and an echelon[2] matrix D (with $d_{11} > 0$) such that $UA = D$. The factoring is done with a process similar to Gaussian elimination. The system $\vec{x}A = \vec{c}$ has an integer solution $\vec{x}$ iff there exists an integer vector $\vec{t}$ such that $\vec{t}D = c$. Since D is an echelon matrix, we can use back substitution to solve for $\vec{t}$ very simply. If no such $\vec{t}$ exists, then the total system is inconsistent. Otherwise, ignoring the bounds, there is a dependence. We then add in the bounds and continue.

If such a $\vec{t}$ exists then the solution $\vec{x}$ is given by $\vec{x} = \vec{t}U$. If the system is not of full rank, and it usually is not, then $\vec{x}$ will have some degrees of freedom. For example: $\vec{t} = (1, t_1)$ $U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ then $\vec{x} = (1, t_1)$ where t_1 (and therefore x_2) can take on any integral value. Wolfe showed that the bounds constraints on $\vec{x}$ can be expressed as constraints on $\vec{t}$ [53]. For example:

```
do i = 1 to 10
  a[i + 10] = a[i]
end do
```

[1] A unimodular matrix is a matrix whose determinant is ± 1.

[2] An echelon matrix is an $m \times n$ matrix such that if the first non-zero element in row i is in column j, then the first non-zero element in row $i + 1$ is in column $k > j$

The initial dependence problem is to find

integers i,i' such that $i + 10 = i'$ and $1 \leq i, i' \leq 10$

The extended GCD Test tells us that $(i, i') = (t_1, t_1 + 10)$. Transforming the constraints to be in terms of t_1 gives us:

$\exists$ integer t_1 such that
$1 \leq t_1 \leq 10$ and
$1 \leq t_1 + 10 \leq 10$

This transformation is valuable for several reasons. First, we have reduced the number of variables. In general, each independent equation will eliminate one variable. Second, we have reduced the number of constraints. Before, each lower bound generated two constraints (one for each array reference), each upper bound generated two constraints and each dimension of the array generated one equality constraint. The equality constraint $a\vec{x} = b$ had to be converted into the two inequality constraints $a\vec{x} \leq b$ and $a\vec{x} \geq b$. Therefore we had $4l + 2d$ constraints (where l is the number of enclosing loops, and d is the number of array dimensions). Now all the equality constraints are folded into the bounds constraints. Thus we are left with only $4l$ constraints.

The complexity of most integer programming algorithms depends on the number of constraints and the number of variables. Thus, in general GCD preprocessing should make the other algorithms more efficient, but more importantly, the form of the new constraints is typically simpler than the original. We have eliminated equality constraints, a necessity for our Acyclic Test discussed below, and we have cut down on the number of variables per constraint. In the previous example, some constraints (the equality ones) contained two variables, while after the transformation all constraints contain one variable. As we shall see, our first exact test, the Single Variable Per Constraint Test, only applies in cases where each constraint has at most one variable.

2.3.2 Single Variable Per Constraint Test (SVPC)

Banerjee shows that if the solution to the generalized GCD Test has at most 1 free variable then one can solve the exact problem easily [8]. Each constraint is merely an

upper or lower bound for the free variable. One merely goes through each constraint calculating the appropriate lower or upper bound and storing the best ones found. If after going through all the constraints, the lower bound is greater than the upper bound, then the test returns "independent". Otherwise the equations are consistent, and the test returns "dependent". Banerjee notes that this can be extended to the case where there are an arbitrary number of variables in the system but at most one variable per constraint. Each constraint is merely an upper or lower bound for one of the free variables. One goes through each constraint and remembers the tightest bound for each variable. If after going through all the constraints, $lb_i > ub_i$ for any variable t_i, the system is inconsistent. Otherwise it is consistent. This test is a superset of the well-known single loop, single dimension exact test [7]. It is quite clear that with one loop and single-dimensional arrays one cannot get constraints with more than one free variable. This test, though, also applies to many common multi-dimensional cases, including those with coupled subscripts, as shown below.

do i_1 = L_1 **to** U_1
  do i_2 = L_2 **to** U_2
    a[i_1][i_2] = a[$i_2 + const_1$][$i_1 + const_2$]
  end do
end do

To demonstrate the algorithm, we cover an instance of the above in detail.

do i_1 = 1 **to** 10
  do i_2 = 1 **to** 10
    a[i_1][i_2] = a[$i_2 + 10$][$i_1 + 9$]
  end do
end do

The GCD Test will set $i_1 = t_1$, $i'_1 = t_2$, $i_2 = t_2 + 9$ and $i'_2 = t_1 - 10$. Expressing the constraints in terms of the t variables we get the following:

$1 \leq t_1 \leq 10$
$1 \leq t_2 \leq 10$
$1 \leq t_2 + 9 \leq 10$
$1 \leq t_1 - 10 \leq 10$

The first constraint sets the lower bound of t_1 to 1 and its upper bound to 10. The second does the same for t_2. The third constraint resets t_2's upper bound to 1. Finally, the last constraint resets t_1's lower bound to 11. Since the lower bound on t_1 is greater than its upper bound, the system is inconsistent.

This algorithm is very efficient. It requires O($C + V$) steps where C is the number of constraints and V is the total number of variables. Each step in the algorithm requires very few operations. Even if certain constraints have more than one variable, applying this test to the applicable ones will eliminate constraints for the following algorithms.

2.3.3 Acyclic Test

We have developed the Acyclic Test for cases where at least one constraint has more than one variable. It works by trying to eliminate variables that are only constrained in one direction. Before applying the Acyclic Test, we first apply the SVPC Test to all the single variable constraints in our system. If any $lb_i > ub_i$, we return independent. Otherwise, we remove all such constraints from further consideration and instead store the lower and the upper bound for each variable that was calculated by the SVPC Test. We then proceed with our Acyclic Test. Given a system of constraints $A\vec{t} \geq 0$ and a variable we wish to eliminate, t_i, we rewrite all our multi-variable constraints to be in the form

$a_{1,i}t_i \leq f_1(t)$
$a_{2,i}t_i \leq f_2(t)$
$\ldots$
$a_{n,i}t_i \leq f_n(t)$

where each $f_j(t)$ is a linear function involving any of the variables except t_i. If every $a_{k,i} \geq 0$, then the variable t_i is only constrained in one direction, to be smaller than

some function of the other variables. Thus we can set t_i to lb_i (the lower bound that the SVPC Test has previously calculated for t_i). If there is a solution with $t_i = lb_i$ then that solution is a solution to the original system. If there is a solution to the original system with $t_i > lb_i$ then clearly setting t_i to the lower value of lb_i will not violate any constraints. Thus there is a solution with $t_i = lb_i$ iff there is a solution to the original system. Note that it is possible that $lb_i = -\infty$ if there is no lower bound for t_i.

Similarly, if every $a_{k,i} \leq 0$, then t_i is only constrained to be larger than some function of the other variables and we can set t_i to its upper bound, $t_i = ub_i$.

Once we set t_i, some constraints may become single variable constraints. We apply the SVPC Test to these constraints. If the SVPC Test can now prove independence, we return independent. Otherwise, we update our bounds, remove these constraints from the system and search for another variable to eliminate.

If we successfully eliminate all the variables without a contradiction, the references are dependent. If we reach a stage where all the remaining variables are constrained in both directions, we say that the Acyclic Test does not apply, and we must try the next test. Even for these cases, the Acyclic Test may be beneficial. Eliminating some of the variables simplifies the system for the next stages.

We illustrate the Acyclic Test with the following example:

$$1 \leq t_1, t_2 \leq 10$$
$$0 \leq t_3 \leq 4$$
$$t_2 \leq t_1$$
$$t_1 \leq t_3 + 4$$

t_1 is constrained in both directions, but we can set t_2 to $lb_2 = 1$. This leaves us with

$$1 \leq t_1 \leq 10$$
$$0 \leq t_3 \leq 4$$
$$1 \leq t_1$$
$$t_1 \leq t_3 + 4$$

Now, t_1 is no longer constrained in both directions. We can set t_1 to $lb_1 = 1$, leaving us with

$0 \leq t_3 \leq 4$
$t_3 \geq 1 - 4$

t_3 can be set to any value between 0 and 4. There are no contradictions so the system is consistent.

As we have described it, the running time complexity of the Acyclic Test is as follows. To find a variable to eliminate, we might have to check every variable to see if it can be eliminated. To decide if a variable can be eliminated, we must check every constraint. Thus to find a variable to eliminate takes O(CV) steps. To actually do the elimination only takes another O(C) steps. To completely solve the system, we must eliminate all the variables so the total complexity in the worst case is O(CV^2).

We can improve this complexity by using a graph algorithm framework. We use the constraints involving more than one variable to create a directed graph. We add two nodes to the graph for each variable t_i; one labeled i and one labeled $-i$. We examine each pair of variables, t_i and t_j, occurring in a single constraint. We first express the constraint as $a_i t_i \leq \ldots + a_j t_j$. If both a_i and a_j are greater than zero, we add an edge to our graph from node i to node j. We then express the constraint as $-a_j t_j \leq \ldots - a_i t_i$ and add an edge from node $-j$ to node $-i$. If a_i is less than zero, we would use node $-i$ for the first edge and node i for the second. Similarly, if a_j is less than zero, we would use node $-j$ for the first edge and node j for the second. Two nodes (one positive, one negative) are needed for each variable to distinguish the case $t_i - t_j + \ldots < 0$ from the case $t_i + t_j + \ldots > 0$. In the first case, t_i is less than t_j plus a function of the other variables while in the second case, t_i is less than $-t_j$ plus a function of the other variables.

Looking at an example, let us assume that we have one constraint $t_1 + 2t_2 - t_3 \leq 0$. We show the resultant graph in Figure 1.

If the resulting graph has no cycles, then we can solve the system exactly using the Acyclic Test. If there is no cycle, there exists a node i (there is no loss of generality in assuming that $i > 0$) such that there are no edges leaving node i (this node is a leaf in the depth-first search tree of the graph). From the method used to construct the graph, this implies that there are no constraints of the form $a_i t_i \leq a_j t_j + \ldots$ where $a_i > 0$. Thus all constraints involving t_i are of the following form:

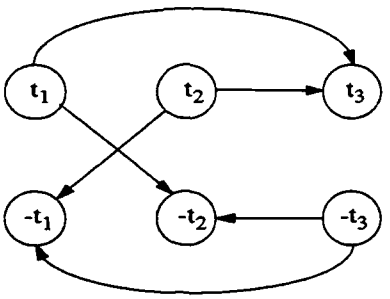


Figure 1: Example graph for Acyclic Test

$$a_{1,i}t_i \geq f_1(t)$$
$$a_{2,i}t_i \geq f_2(t)$$
$$\ldots$$
$$a_{n,i}t_i \geq f_n(t)$$

where $f(t)$ is a linear function involving any of the variables except t_i and where $a_{1,i} > 0, \ldots, a_{n,i} > 0$. From our previous discussion, we can eliminate variable t_i by setting it to ub_i.

If our initial node was $-i$ rather than i, then all constraints on t_i would be in the other direction, i.e. $a_{1,i} < 0, \ldots, a_{n,i} < 0$ and we would set t_i to lb_i.

Once we set t_i, the next node visited in on our depth-first search must also be constrained in only one direction, and we can set it. We continue until we reach a contradiction, a lower bound larger than an upper one, or until no variables are left. If we eliminate all the variables without finding a contradiction, then the system is consistent. Otherwise it is inconsistent.

This version of the algorithm can be somewhat more efficient since we do not have to check all the variables when searching for a variable to eliminate. Its complexity is proportional to the number of edges in the graph. If each constraint contains a term for

every variable, we will have O(CV^2) edges in the graph, but in the sparse systems we see in practice, we expect it to be much smaller.

For the small systems we see in practice, the better asymptotic complexity might not make much of a difference. For simplicity, we have chosen to only implement the non-graph version of the algorithm.

The Acyclic Test illustrates the benefit of using the GCD Test as a preprocessing step to eliminate equality constraints. For example, without GCD preprocessing, the simple equality constraint $i_1 = i_2$ would be represented as the two constraint $i_1 \leq i_2$ and $i_1 \geq i_2$. These two constraints alone create a cycle in the graph ($i_1 \leq i_2 \leq i_1$).

2.3.4 Simple Loop Residue Test

If there is a cycle in our graph, we attempt the Simple Loop Residue Test. Pratt developed a simple algorithm that can be used for data dependence testing that works when all constraints are of the form $t_i \leq t_j + c$ [39]. One creates a graph with a node for each variable. For this inequality, we place a directed arc with value c from node t_i to node t_j. Assume we have another constraint $t_j \leq t_k + d$. By transitivity, this implies that $t_i \leq t_k + c + d$. We define the value of a path in the graph to be equal to the sum of the values of the edges on the path. In the above example, the value of the path from node i to node k is $c + d$. So the value of the path constrains its endpoints in the same way that an edge does. Thus, if there is a path from node n_1 to n_2 with value v, we know that $n_1 \leq n_2 + v$. Constraints with only one variable are also acceptable. We create a special node, n_0. The constraint $t_i \leq c$ is represented with an edge from i to n_0 with value c. Similarly, the constraint $t_i \geq c$ is represented with an edge from n_0 to i with value $-c$. A cycle in the graph represents a constraint of the form $i - i \leq c$ or $0 \leq c$. We check every cycle in the graph. If any cycle has a negative value, the system is inconsistent. Otherwise it is consistent. While a graph may have an exponential number of cycles, checking if a graph has a negative cycle can be done in time proportional to the number of edges times the number of vertices using Bellman-Ford's algorithm [48]. Since our graph has one vertex for every variable and one edge for every constraint, the complexity of the Simple Loop Residue Test is O(CV).

Shostak [46] extends this algorithm first to deal with inequalities of the form $at_i \leq bt_j + c$ and then to handle cases with more than two variables.[3] Unfortunately these extensions make the algorithm inexact. However, the algorithm can be extended to the case $at_i \leq at_j + c$ without losing exactness. This case is equivalent to $a(t_i - t_j) \leq c$. Let d be the largest integer such that $d \leq c$ and d is a multiple of a. We can replace the inequality with $t_i - t_j \leq d/a$ where d/a is an integer.

As an example of the Simple Loop Residue Test, assume we have the following constraints:

$1 \leq t_1, t_2 \leq 10$
$0 \leq t_3 \leq 4$
$t_2 \leq t_1$
$2t_1 \leq 2t_3 - 7$

Figure 2 shows the graph after converting the last constraint to $t_1 \leq t_3 - 4$.

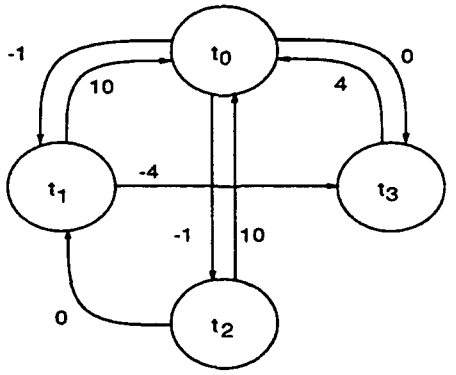


Figure 2: Example graph for Loop Residue Test

There is a cycle from t_1 to t_3 to t_0 to t_1 with value $-4 + 4 - 1 = -1$. Therefore the system is inconsistent.

[3]For clarity we use the term Simple Loop Residue Test to refer to the Loop Residue Test without Shostak's extensions.

2.3.5 Fourier-Motzkin Test

Our last algorithm, Fourier-Motzkin [12], is a backup inexact test. It solves the general non-integer linear programming case exactly. If it returns independent, we know that the integer case is also independent. If it returns dependent, we can use it to return a sample solution. If this sample solution is integral, then the integral case is dependent. Otherwise we are not sure. In the few cases where our first four algorithms did not apply, and we were required to call Fourier-Motzkin, the algorithm was always exact (i.e., it either returned independent or the sample solution was integral). The cost of this algorithm is a matter of debate. Theoretically it can be exponential. Experimentally, Triolet has implemented this approach and seems to be satisfied with its efficiency [51], but Li, Yew and Zhu consider Triolet's numbers to be too expensive [32]. Nonetheless, we are required to call this algorithm so few times that its accrued expense is very reasonable.

In the first step, we eliminate the first variable, x_1, from the set of constraints. All the constraints are first normalized so that their coefficient for x_1 is 0, 1 or -1. The set of constraints is then partitioned into three sets depending on the value of the coefficient.

$$x_1 \geq D_1(\vec{x}_{2,\ldots,n}) \ , \ldots, \ x_1 \geq D_p(\vec{x}_{2,\ldots,n})$$
$$x_1 \leq E_1(\vec{x}_{2,\ldots,n}) \ , \ldots, \ x_1 \leq E_q(\vec{x}_{2,\ldots,n})$$
$$0 \leq F_1(\vec{x}_{2,\ldots,n}) \ , \ldots, \ 0 \leq F_r(\vec{x}_{2,\ldots,n})$$

This system has a solution iff

$$\exists \vec{x}_{2,\ldots,n} \text{ such that}$$
$$D_i(\vec{x}_{2,\ldots,n}) \leq E_j(\vec{x}_{2,\ldots,n}), \ i = (1,\ldots,p), \ j = (1,\ldots,q)$$
$$0 \leq F_k(\vec{x}_{2,\ldots,n}), \ k = (1,\ldots,r)$$

A proof can be found in [12]. Thus, one can eliminate one variable at a time until there are none left. At each step, the number of constraints grows by $pq - p - q$. While this can lead to exponential behavior in the worst case, when p and q are small, each step might actually eliminate constraints. If there is a solution $\vec{x}_{2,\ldots,n}$ then the original system

will be satisfied with any x_1 such that $\max(D(\vec{x}_{2,\ldots,n})) \leq x_1 \leq \min(E(\vec{x}_{2,\ldots,n}))$. Thus one can back substitute to find sample solutions for all the x variables. We would like the sample solution to be integral. As a heuristic, at each step of the back substitution, we set x_i to be the integer at the middle of the allowed range.

We have extended the Fourier-Motzkin algorithm in two simple ways. This allows us to sometimes be able to prove independence even when there are real solutions. When we are computing our sample solution, at each step i in the back substitution, we know that there exists a real solution for x_i such that $\max(D(\vec{x}_{i+1,\ldots,n})) \leq x_i \leq \min(E(\vec{x}_{i+1,\ldots,n}))$, where we have already substituted in values for $\vec{x}_{i+1,\ldots,n}$. Having no integer in the allowable range for x_i does not necessarily imply that there is no integer solution. If we had substituted a different sample value for some earlier $\vec{x}_{i+1\ldots n}$, we might have found an integral solution.

Suppose, though, that there is no allowable integer for our first variable, x_n. Since we have not constrained ourselves by selecting a value for any other x_j, we can be assured that there is no integer solution, and the system *is inconsistent.*

A second extension involves *implicit* branch and bound. Most systems, including ours, do not just test pairs of references for independence. They also compute the distance and direction vectors between the pairs. Recall that given a dependent iteration pair $(\vec{i},\vec{i'})$, the distance vector between the two references is $\vec{d} = \vec{i'} - \vec{i}$ and the direction vector $\vec{s}$ is the sign of the distance vector, $\vec{s} = (\text{sign}(d_1), \text{sign}(d_2), \ldots)$. Since all iteration values are integral, we use directions to partition a distance into one of three sets; $d_i = 0$, $d_i \geq 1$, $d_i \leq 1$. Thus if there is a real dependence with a fractional distance $0 < d_i < 1$, Fourier-Motzkin might say that the two references are dependent but have no dependent direction vectors. In such a case, we know that the two references are in fact independent.

In general, if the sample solution is not integral, one must use full integer programming techniques such as branch and bound. Say for example that $x_i = 3.5$. Then one sets up two companion systems. One with the added constraint that $x_i \leq 3$ and another with the constraint $x_i \geq 4$. If neither system has a solution then the original system is inconsistent. It is possible that after this step one is still left with a non-integral solution. One can then repeat the branch and bound step. Conceivably, one might be required to branch and bound many times (proportional to the size of the region). In such a case,

one might have to cut off the process after an arbitrary number of steps and assume dependence. We have not found any cases that require us to use explicit branch and bound.

2.4 Effectiveness of Algorithms

We have implemented these algorithms in the SUIF system [49], a compiler system developed at Stanford. We then ran them on the PERFECT Club benchmarks. These are a set of 13 scientific Fortran programs ranging in size from 500 to 18,000 lines collected at the University of Illinois at Urbana-Champaign [14]. We feel that these are a fair set of benchmarks. Non-scientific codes do not exhibit much loop level parallelism, and we feel that library routines tend to be simpler than full programs.

Table 1 shows how many times each test was called per program. The first column represents array constants, for example a[3] versus a[4]. These cases are handled without dependence testing. We merely include them to show that their frequency can skew statistics if we apply general dependence routines to them. The second column represents the cases where GCD returns independent. For these cases, we do not need to call the exact routines. The other columns correspond to the number of successful applications of the tests. There are two key results. First, the vast majority of cases are handled with the efficient Single Variable Per Constraint Test. Second, there is an implicit final column for the cases when Fourier-Motzkin failed. Since this never happened, the column of 0's is eliminated. In no cases were we required to use a general integer programming algorithm such as branch and bound.

2.5 Memoization To Improve Efficiency

We have shown that our algorithms are exact in every instance of our benchmarks. Now we consider efficiency. We have argued that using special case exact tests is inherently efficient since most problems can be solved using only one test. We can further improve the efficiency of our approach. It has been said that dependence testing is performing a large number of tests on relatively small inputs [32]. We show that in actuality it is

Table 1: Number of times each test was called for each program in the PERFECT Club

Affine Memory Disambiguation Test Frequency							
Program	#Lines	Constant	GCD	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	229	91	613	0	0	0
CS	18,520	50	0	127	15	0	0
LG	2,327	6,961	0	73	0	0	0
LW	1,237	54	0	34	43	0	0
MT	3,785	49	0	326	0	0	0
NA	3,976	45	0	679	202	1	2
OC	2,739	2	7	36	0	0	0
SD	7,607	949	0	526	17	5	12
SM	2,759	1,004	98	264	0	0	0
SR	3,970	1,679	0	1,290	0	0	0
TF	2,020	801	6	826	0	0	0
TI	484	0	0	4	42	0	0
WS	3,884	36	182	378	4	0	160
TOTAL	59,412	11,859	384	5,176	323	6	174

performing a small number of unique tests on small inputs repeatedly. There is little variation in array reference patterns found in real programs, and most bounds tend to go from 1 to n where n is the same throughout the program. Thus, one can save much computation by using memoization,[4] remembering the results of previous tests. The use of a hash table allows us to find a duplicate call very quickly.

A simple memoization scheme does not repeat a test if the input exactly matches a previous one. We can improve the effectiveness of memoization by eliminating the loop bound constraints of loop indices that are unused by the array reference function. If a loop index is unused by either reference function, its bounds do not affect the dependence of the references. Eliminating such loops increases the likelihood that different pairs will match in the memoization table. Both of the following programs

```
(a) do i = 1 to 10
      do j = 1 to 10
        a[i+10] = a[i]+3
      end do
    end do
```

```
(b) do i = 1 to 10
      do j = 1 to 20
        a[i+10] = a[i]+3
      end do
    end do
```

collapse to this one:

```
do i = 1 to 10
  a[i+10] = a[i]+3
end do
```

[4]Memoization is a technique to remember the results of previous computations, previously used to implement call-by-need arguments in LISP compilers [1].

Two cases that before appeared to be different, now are identical.

In Table 2 we show the results of both our simple scheme and our improved one applied to the PERFECT Club. We use two hash tables; one using loop bounds and one not. The GCD Test does not make use of bounds. Thus, if a particular reference matches ignoring the bounds, we are not required to repeat the GCD Test.

Table 2: Percentage of unique cases for memoization schemes

Percentage of Unique Cases						
Program	Without Bounds (GCD)			With Bounds		
	Total	Unique		Total	Unique	
		Simple	Improved		Simple	Improved
AP	704	7.0%	4.4%	613	6.4%	4.4%
CS	142	7.7%	7.0%	142	16.2%	14.1%
LC	73	32.9%	13.7%	73	47.9%	31.5%
LW	77	11.7%	10.4%	77	23.4%	22.1%
MT	326	3.4%	2.5%	326	6.4%	4.3%
NA	884	4.2%	3.4%	884	7.9%	6.9%
OC	43	27.9%	20.9%	36	19.4%	13.9%
SD	560	6.6%	6.1%	560	9.5%	8.8%
SM	362	5.5%	3.6%	264	4.9%	3.0%
SR	1,290	1.1%	0.9%	1,290	1.6%	1.1%
TF	832	2.2%	1.7%	826	2.9%	2.4%
TI	46	30.4%	19.6%	46	34.8%	23.9%
WS	724	11.9%	11.0%	542	14.2%	11.6%
TOT	6,063	5.7%	4.4%	5,679	7.3%	5.8%

In Table 3 we show how memoization improves the results of Table 1. The Total Cases column gives the total number of exact tests from Table 1. The remaining columns show how many of each exact test is left after memoization. Memoization reduces the total from 5,679 to 332 tests!

Further optimizations are possible. For example, one can eliminate symmetrical cases. Assume there are two references in a loop; r_1 and r_2. We wish to know if this case is equivalent to a pair of references in our table; r'_1 and r'_2. Assume the bounds are the same. Our simple scheme would say the two cases are equivalent iff $r_1 = r'_1$ and $r_2 = r'_2$,

Table 3: Number of times each test was called looking only at unique cases

Dependence Test Frequency For Unique Cases						
Program	#Lines	Total Cases	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	613	27	0	0	0
CS	18,520	142	14	6	0	0
LG	2,327	73	23	0	0	0
LW	1,237	77	15	2	0	0
MT	3,785	326	14	0	0	0
NA	3,976	884	48	11	1	1
OC	2,739	36	5	0	0	0
SD	7,607	560	36	6	3	4
SM	2,759	264	8	0	0	0
SR	3,970	1,290	14	0	0	0
TF	2,020	826	20	0	0	0
TI	484	46	3	8	0	0
WS	3,884	542	35	1	0	27
TOTAL	59,412	5,679	262	34	4	32

but the cases are actually also equivalent if $r_1 = r_2'$ and $r_2 = r_1'$. For example comparing a[i] to a[$i-1$] is the same as comparing a[$i-1$] to a[i]. This can be taken farther: a[i][j] versus a[$i+1$][$j+1$] is equivalent to a[j][i] versus a[$j+1$][$i+1$].

One other possible improvement is to store the hash table across compilations. This will eliminate the data dependence cost of incremental compilation. In addition, if there is similarity across programs, one could use a set of benchmarks to set up a standard table that would be used by all programs.

Our hash table is implemented using a simple-minded open table hashing scheme. Treating the input data, array reference equations and loop bounds, as one long vector, $\vec{x}$, of integers, our hashing function is $h(\vec{x}) = size(\vec{x}) + \sum_i 2^i x_i$. This function was chosen so that symmetrical or partially symmetrical references would not collide. Because of the low number of unique cases, random collisions are not much of a problem.

Our hashing scheme performs well enough for our purposes. The hit cost does not appear to be too high. If necessary, though, we believe that we could do much better. Our hashing table is applied to a pair of references (r_1,r_2). We can use another hash

table to assign each array reference in the program a reference number. Two references would have the same reference number iff they were equivalent. Since there are far fewer references than pairs of references, this would be very inexpensive. We could then define a new hashing function for pairs that would be some function of the two reference numbers. Since the size of our input data, $\vec{x}$, is typically much larger than two integers, computing this hashing function could be even cheaper than scanning the input data.

2.6 Direction and Distance Vectors

Typically, we are not just interested in knowing if two references are independent. We must also compute the distance and direction vectors for the pair.

Two references are dependent with distance $\vec{d}$ if there exists a dependent iteration pair, (i,i'), such that $\vec{i'} - \vec{i} = \vec{d}$. The extended GCD Test, by computing an expression for the dependent iteration pairs, provides us with an easy way to compute distance vectors. For example:

```
do i = 0 to 10
  a[i] = a[i-3]+7
end do
```

The extended GCD Test tells us that for all dependent iteration pairs (i,i'), $i = t_1$ and $i' = t_1 + 3$. We merely subtract these two expressions to compute the distance vector $d = (3)$. This method does not work when the two references are dependent with more than one distance vector. In such cases, subtracting the t expressions would result in a non-constant vector. Typically, though, we are less interested in non-constant distances.

In addition, since the extended GCD Test does not use bounds, this method does not work for cases where the distance is only constant because of the bounds. For example:

```
do i = 1 to 8
  do j = 1 to 10
    a[10*i+j] = a[10*(i+2)+j]+7
  end do
end do
```

We will not discover that the only distance vector is (-2,0). Nonetheless, these types of examples are fairly rare and the extended GCD Test should be sufficient for the common constant-distance cases.

Pugh has developed a more accurate method for computing distance vectors [40]. For each pair of loop indices, (i_j,i'_j), he adds to his system of constraints a new variable d_j and the constraint that $d_j = i'_j - i_j$. Then, using an extension to the Fourier-Motzkin algorithm, he eliminates all the variables except the $\vec{d}$ variables. Using Fourier-Motzkin to eliminate a variable is equivalent to projecting the system of constraints to the space of the remaining variables. Thus after eliminating all the other variables, Pugh is left with a series of constraints on the $\vec{d}$ variables that describes all the dependent distance vectors. While Pugh's method is more accurate, it is also more expensive. Adding extra variables and constraints to his system increases the cost of his test.

Not all dependent references are dependent with a constant distance vector. When a pair of references is dependent with many distances, it is useful to use direction vectors to summarize all these dependences. While a pair of references may be dependent with more than one direction vector, the number of dependent direction vectors can be much smaller than the number of dependent distance vectors. For example:

```
do i = 0 to 10
  do j = 0 to 10
    a[i][j] = a[2*i][j]+7
  end do
end do
```

These two references are dependent with the dependent iteration pairs $((i_w, j_w), (i_r, j_r)) \in \{((0, j),(0, j)), ((2, j),(1, j)), ((4, j),(2, j)), ((6, j),(3, j)), ((8, j),(4, j)), ((10, j),(5, j))\}$ where $j = 0 \ldots 10$. The two references are dependent with the two direction vectors (-, 0) and (0, 0), but the six distance vectors (0, 0), (-1, 0), (-2, 0), (-3, 0), (-4, 0) and (-5, 0). There is little benefit in this case to the extra precision of distance vectors.

A simple method to compute direction vectors is to enumerate all the possible direction vectors[5] and ask if the references are independent subject to the current vector. Each direction vector is a set of simple linear constraints on the loop indices. We simply add these constraints to our system and solve as before. The standard approach, based on Burke and Cytron [10], uses a hierarchical system to avoid testing all possible direction vectors. Rather than testing each possible vector it first tests $(*, *, \ldots, *)$. Recall that $*$ represents a dependence with any direction. If this returns independent, we know there are no direction vectors for which the references are dependent. If it returns dependent, we then preform the tests with each of the following vectors $(+, *, \ldots, *)$, $(0, *, \ldots, *)$ and $(-, *, \ldots, *)$. If, for example, the first vector returns independent we know that there is no dependence with any vector whose first component is $+$. We can prune any such vector. If any vector returns dependent, we continue to expand its $*'s$.

The addition of direction vector constraints can conceivably limit the applicability of our tests. For example:

$$1 \leq t_1, t_2, \leq 10$$
$$t_1 \leq t_2$$

can be solved with the Acyclic Test. A direction vector may add the constraint $t_2 < t_1$. The Acyclic Test is no longer applicable and we must use the Loop Residue Test. Similarly, there could be cases where direction vectors force us to use the Fourier-Motzkin Test. In practice, we have observed a greater need for Acyclic and Loop Residue, but in no case have we been forced to use Fourier-Motzkin due to the addition of the extra constraints.

A more serious problem is that direction vectors require the dependence tests to be applied multiple times for a pair of references. The number of possible vectors is

[5]Note, with distance vectors we would have had to enumerate all possible dependences. With direction vectors we only need to enumerate all possible directions, a large but much smaller number.

potentially exponential in the loop nesting. Even using hierarchical pruning, without further optimizations we still have problems in practice. In Table 4, we repeat Table 3 with direction vectors, counting every direction tested. This overestimates the cost since certain fixed costs such as the GCD Test and the overhead of transforming the bounds to be in terms of the t variables do not depend on how many vectors are tried per test.

Table 4: Number of times each test was called computing direction vectors

Dependence Test Frequency For Direction Vectors					
Program	#Lines	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	363	104	100	0
CS	18,520	127	48	34	0
LG	2,327	1,067	1,138	4,619	0
LW	1,237	132	73	59	0
MT	3,785	120	32	16	0
NA	3,976	295	124	172	23
OC	2,739	37	8	4	0
SD	7,607	309	106	120	28
SM	2,759	355	110	169	0
SR	3,970	130	30	18	0
TF	2,020	169	16	11	0
TI	484	780	267	703	0
WS	3,884	303	105	52	106
TOTAL	59,412	4,187	2,161	6,077	157

Even allowing for a generous overestimate, calculating direction vectors has greatly increased the number of tests performed. Before, the compiler called 332, mostly SVPC, tests. Now, it needs to call about 12,500, mostly Acyclic and Loop Residue tests. The number of times Fourier-Motzkin is applied has gone from 32 to 157 (note that this is solely due to checking multiple vectors for the same references).

Some simple pruning methods can bring these costs back down dramatically. In discussing memoization, we mentioned that we need not include unused variables. For example:

```
do i = 1 to 10
  do j = 1 to 10
    a[j] = a[j + 1]
  end do
end do
```

Since i does not appear in neither the array expression nor in a loop bound, we know that direction for $i' - i$ is $*$. Thus we run the tests for j and then prepend a $*$ to the resultant direction vectors.

Calculating distance vectors can also help. If, for example, we have

```
do i = ...
  a[i + 1] = a[i]
end do
```

we know from the GCD Test that $i' - i = 1$. We therefore know that $i' > i$ and need not try out any other directions.

Table 5 shows our results with unused variables eliminated and with distance vector pruning.

We now have to call the tests only about 900 times. If we need better results, Burke and Cytron suggest as an optimization that *nice* cases can be treated on a dimension by dimension basis rather than as a system. For example:

```
do i = 1 to 10
  a[i + 1][j] = a[i][j]
end do
```

i and j are not interrelated and we can compute each component of the direction vector independently.

Table 5: Number of times each test was called using pruning

Unused Variables and Distance Vector Pruning					
Program	#Lines	SVPE	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	27	6	6	0
CS	18,520	14	16	14	0
LG	2,327	44	6	6	0
LW	1,237	15	12	5	0
MT	3,785	14	0	0	0
NA	3,976	48	59	118	7
OC	2,739	5	0	0	0
SD	7,607	54	20	55	28
SM	2,759	8	0	0	0
SR	3,970	14	0	0	0
TF	2,020	23	0	0	0
TI	484	3	38	72	0
WS	3,88 4	35	15	0	106
TOTAL	59,412	304	172	276	141

2.7 Cost of Affine Memory Disambiguation Tests

In the previous sections, we have suggested that our suite, combined with memoization, can solve the affine memory disambiguation problem exactly and efficiently. In this section, we explicitly show the efficiency of our approach. In Table 6 we timed our dependence tests and compared them to standard scalar optimizing compilers (f77 -O3). The entries labeled * were too small to measure. We wish to show that being exact adds very little cost to compilation time. The timings do not include the set up time required for dependence testing, for example expressing a reference in terms of the loop indices. This setup time, while possibly significant, is equivalent for all methods. The timings therefore should be looked upon as an upper bound for the extra time required to use our approach. The standard compilation time used full scalar optimizations. Our approach added only about 3% on average to the compile time.

Table 6: Total cost of affine memory disambiguation testing

Dependence Testing Cost		
Program	Dep. Test Cost (in secs)	f77 -O3 (in secs)
AP	2.2	151.4
CS	*	485.0
LG	4.0	65.4
LW	1.1	33.0
MT	1.0	45.0
NA	3.6	136.3
OC	0.3	38.2
SD	2.7	62.1
SM	3.5	102.5
SR	3.8	118.5
TF	2.6	116.6
TI	0.7	12.6
WS	3.6	110.0

2.8 Discussion

We have shown that our algorithms can be exact and efficient in practice. We have not shown how being exact compares with other, inexact, approaches. Looking at pairs of references does not give the entire picture. In a loop with a thousand independent pairs, being inexact in just one test could have a devastating effect on the amount of parallelism discovered. Ideally, one would like a standard model to measure the parallelism found. Then one could say how much faster a program ran due to exact affine memory disambiguation. Unfortunately it is very difficult to define such a model since the amount of parallelism depends on the optimizations performed.

Nonetheless, to give some comparison, we implemented two standard algorithms: the Simple GCD Test (algorithm 5.4.1 in [8]) and the Trapezoidal Banerjee Test (algorithm 4.3.1 in [8]). Not computing direction vectors, these algorithms found 415 out of 482 independent pairs, missing 16%. For direction vectors, we used the Simple GCD Test followed by Wolfe's extension to Banerjee's rectangular test (2.5.2 in [53]). We eliminated unused variables so that a[i] versus a[i-1] would return the one direction vector $(*, +)$

and not, for example, the three direction vectors $((+,+)\ (0,+)\ (-,+))$ that would be returned if these references were also enclosed by a second, unused outer loop. These algorithms returned 8,314 direction vectors, which is 22% more than the exact answer of 6,828.

We do not believe that our particular choice of tests is very important. Our key contribution is that it is possible to be exact and efficient in practice using a suite of special case exact tests. It is quite possible that other tests could be added and some eliminated without significantly changing our results. In fact, since we first presented these results, Pugh has come up with an alternative exact approach [40]. He uses a variation of the Extended GCD Test followed by an extension to the Fourier-Motzkin algorithm. His approach may be a little more expensive than ours. He claims that his test requires $O(CV^2)$ worst case time for the common cases when the SVPC Test applies, while we have shown that the SVPC Test requires $O(C+V)$ time. Pugh does not believe in memoization [40]. He makes the incorrect claim that the cost of a memoization hit would be about 2 to 4 times the scanning, or copying, cost of a problem. We have shown that in fact the hit cost could be less than the scanning cost. Nonetheless, we believe that with memoization Pugh's approach might not be much more expensive than ours.

We included the Extended GCD Test in our suite because it increases the applicability of the other tests. We chose our other tests because of their efficiency, applicability and compatibility. They all expect their data in the same form: $A\vec{x} \leq \vec{b}$. Thus there is no need to convert data from one form to another. Some tests, like the Lambda Test [32] expect their data in a different form. All the tests succeed in finding independent references in practice. We checked how many times each test returned independent (counting each direction vector tested) for the tests in Table 5. SVPC returned independent in 40 out of 308 cases, Acyclic in 14 out of 172 cases, Loop Residue in 131 out of 276 cases and Fourier-Motzkin in 82 out of 141 cases. We ordered the tests by cost. Thus, in most cases, we only need to call our most efficient test, SVPC.

2.9 Extension to Symbolic Testing

Our tests expect all references and bounds to be linear functions of the loop indices. We mentioned before that we use optimization techniques (constant propagation and induction variable and forward substitution) to increase the applicability of these conditions [49][50].

Nonetheless, there are cases where the unknown symbolics variables cannot be expressed as functions of the loop indices.

```
read(n)
...
do i = 1 to 10
  a[i+n] = a[i+2*n+1]+3
end do
```

As long as we know that n does not vary inside the loop, we can add it to our system as if it were an induction variable without bounds. For this example, our system would ask the following question.

$\exists$ integers i, i' and n such that
$1 \leq i,\ i' \leq 10$ and
$i + n = i' + 2 * n + 1$

Table 7 shows the results of adding symbolic testing to our system. Our tests are now called about 1,060 times compared with about 900 times before. This should add little to our total cost. We speculate that the low cost is because our prepass optimizations are quite powerful.

Other groups have advocated more powerful symbolic techniques [21][33]. In the next chapter, we will describe these other approaches in greater detail and argue that our approach is sufficient.

Table 7: Number of times each test was called computing direction vectors and adding symbolic constraints

Symbolic Testing					
Program	#Lines	SVPE	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	33	22	6	0
CS	18,520	20	24	19	0
LG	2,327	48	6	6	0
LW	1,237	15	12	5	0
MT	3,785	19	0	0	0
NA	3,976	55	149	101	7
OC	2,739	5	1	0	0
SD	7,607	54	20	55	28
SM	2,759	8	0	0	0
SR	3,970	21	1	2	0
TF	2,020	43	0	0	0
TI	484	3	38	72	0
WS	3,884	35	19	0	106
TOTAL	59,412	359	292	266	141

2.10 Chapter Conclusions

Affine memory disambiguation is a fundamental component in any parallelizing compiler. Previous techniques have required approximations. We have presented a definitive solution to the problem by using a combination of simple, easy to implement techniques; cascading special case exact tests, memoization and better direction vector pruning. In practice, our tests have found independent references that could not be found with currently used techniques. Testing on large benchmarks, we demonstrate empirically that our method is both exact and inexpensive.

Chapter 3

Effectiveness of Affine Memory Disambiguation

We have shown in the previous chapter that intra-procedural affine memory disambiguation can be solved exactly and efficiently. However, we want to solve data dependence in general and not just affine memory disambiguation. Compilers restrict themselves to this domain because it is easier to solve. The question remains how accurately this domain approximates the full data dependence problem.

Solving the affine memory disambiguation problem exactly allows us to judge the effectiveness of the domain restriction. Using a dynamic, trace-based system developed by Larus [28], we can find a solution to the full data dependence problem for a given input set. By comparing our exact affine memory disambiguator to Larus's system, we can discover the limitations of the domain. We show that while the affine approximation is reasonable, the memory disambiguation approximation is not. This experiment was first described in [38].

3.1 Experimental System

In the last chapter we described our dependence analysis system. It solves exactly the affine memory disambiguation problem with an extension for symbolic analysis.

We use llpp, a system developed by Larus, to dynamically find all the loop-carried

true data-flow dependences in a program for a given input set [28]. Recall that a loop-carried true data-flow dependence occurs when we read a value in iteration i_r that was written in an earlier iteration i_w. Larus developed his system to dynamically estimate the amount of loop level parallelism in a program. It modifies a standard compiler, gcc, to add tracing code to the sequential program. By keeping track of high level information such as the current loop iteration and the loop iteration when a value was written, it is able at each read to determine if there is a loop-carried true data-flow dependence. Llpp gives us the exact number of true data-flow dependences for a given input. Because it is a dynamic system, it need never assume that a true data-flow dependence exists when one does not, but it will not find dependences that could occur with a different input set.

The scope of our study is limited. Looking at pairs of references does not necessarily give one an accurate picture. If a particular program has 1,000 array reference pairs, all of which are independent, and our system is able to prove that 999 are independent, we would say that our system was accurate in 99.9% of the cases. Nonetheless, we would have eliminated all the potential parallelism. A better study would discover how much parallelism was lost due to the domain restrictions. This would require an agreed upon model of the amount of parallelism in particular programs. This model may be very complex. It could depend on the input data and the target machine. Unfortunately, such a model does not yet exist.

Our study ignores all true data-flow dependences where the read and write are in different subroutines. We do not count in our statistics any interprocedural array pairs that according to Larus have loop-carried true data-flow dependences. To prove two such references independent requires interprocedural data dependence analysis. Chapter 5 discusses interprocedural analysis in detail.

We ignore anti and output dependences. While they do not inherently limit parallelism, they do need to be recognized to be eliminated. Our compiler system is capable of differentiating different types of dependences. We do not believe that it is any more difficult for our system to detect these types of dependences than it is for our system to detect true dependences, so we do not believe that their exclusion in the experiment significantly affects our results.

We look at five programs in the PERFECT Club, ranging in size from 500 to 7,600

lines. Looking at five programs does not constitute a broad enough study. We were forced to limit ourselves mainly for two reasons. First, llpp requires a very large amount of memory on larger traces. Second, it was not possible to automate the categorization of the failure cases; we had to compare the results of the two systems by hand. While we believe that a deficiency that is widely found in these five programs is probably a real problem, the reverse may not be true. There could be other limitations of affine memory disambiguators that do not appear in these benchmarks.

3.2 Experimental Results

Table 8 shows the results of applying llpp to each of the programs in our suite. We compared every array write statement with every array read statement. The first column shows how many of the array reference pairs have loop-carried true data-flow dependences for the given input data (We used the standard input data that comes with the PERFECT Club). The second column shows how many of the pairs have no loop-carried true data-flow dependences. The large majority of the cases have no loop-carried true data-flow dependences. Thus we can be hopeful that these programs have some inherent parallelism.

Table 8: Number of loop-carried true data-flow dependences for each program

Number of Loop-Carried True Data-flow Dependences			
Program	#Lines	True Data-flow Dependence	No True Data-flow Dependence
LG	2,327	64	6,006
LW	1,237	28	690
SD	7,607	21	379
SR	3,970	41	1,654
TI	484	16	36
TOTAL	15,625	170	8,765

Table 9 shows the effectiveness of affine memory disambiguation. As any conservative compilers will err on the side of assuming dependences, all systems should correctly

handle the cases that do have loop-carried true data-flow dependences. We therefore only compare the 8,765 pairs that do not have loop-carried true data-flow dependences.

Table 9: Effectiveness of restricting domain to affine memory disambiguation

Effectiveness of Domain Restrictions				
Program	#Lines	Successes	Affine Failures	Disambiguation Failures
LG	2,327	5,485	0	521
LW	1,237	290	5	395
SD	7,607	207	19	153
SR	3,970	1,475	23	156
TI	484	0	0	36
TOTAL	15,625	7,457	47	1,261

The first column shows the number of pairs that could be resolved correctly, i.e. proven by our system to not have any loop-carried true data-flow dependences. The affine memory disambiguation domain is sufficient to correctly solve about 85% of the cases. We examined each of the failure cases by hand to determine which domain restriction was responsible for the failure. In the second column, we show the number of cases that fail due to the affine restrictions. In the last column, we show the number that fail due to memory disambiguation failures. While we have very few failures due to the affine restrictions, solving only the memory disambiguation problem leads to many errors. In the next section, we will categorize the types of failures in detail.

3.3 Memory Disambiguation Failures

Most of our failures were due to the memory disambiguation restriction. Table 10 divides these failures into three categories that we will discuss in detail.

3.3.1 Data-flow

The largest category of failures is what we classify as data-flow failures. These are the types of examples we discussed in Chapter 1.

Table 10: Categorization of memory disambiguation failures

Memory Disambiguation Failures				
Program	#Lines	Data-flow	Dynamic	Harmless
LG	2,327	258	17	246
LW	1,237	379	0	16
SD	7,607	27	51	70
SR	3,970	93	0	63
TI	484	10	0	26
TOTAL	15,625	767	68	421

```
do i = 1 to n
  do j = 1 to m
    work[j] = ...
  end do
  do j = 1 to m
    ...= work[j]
  end do
end do
```

There is no loop-carried true data-flow dependence between the two array statements in the above example. In each iteration of the outer loop, the work array is being reinitialized and then used. While the location being read in every iteration of the outer loop, i_r, was written in all the previous iterations, the value read was written in the same iteration, $i_w = i_r$. It is impossible to discover this looking only at pairs of references. For example:

```
do i = 1 to n
  if i < 4 then
    do j = 1 to m
      work[j] = ...
    end do
  end if
```

```
  do j = 1 to m
    ... = work[j]
  end do
end do
```

The two array references are identical to the ones in the first example, but in this case there is a loop-carried true data-flow dependence. The values being read in iterations $i = 4$ to $i = n$ were all written in iteration $i = 3$. To differentiate the two cases requires data-flow information.

Even if one can differentiate the two cases, the first case cannot simply be run in parallel. There is an output dependence that has to eliminated. This can be accomplished using array privatization or array expansion. Each processor can be given its own private copy of the work array. Each processor would then run the following code where `mywork` is the processor's local copy of the array:

```
do i = myfirst to mylast
  do j = 1 to m
    mywork[j] = ...
  end do
  do j = 1 to m
    ... = mywork[j]
  end do
end do
```

The majority of our total failures were due to data-flow problems. This indicates that array privatization is very important. We discuss data-flow dependences and array privatization in detail in Chapter 4.

3.3.2 Dynamic

Another memory disambiguation problem relates to dynamic versus static dependences. For example:

```
read(n)
do i
  if n == 0 then
    a[i] = ...
  end if
  ... = a[i - 1]
end do
```

Assume n is never zero for the input set we use. There is no true data-flow dependence as the write statement is never executed. Clearly the compiler cannot assume that n is never zero. What might be useful is for the compiler to use versioning. By hoisting the conditional out of the loop, we create two versions of the loop as follows:

```
read(n)
if n == 0 then
  do i
    a[i] = ...
    ... = a[i - 1]
  end do
else
  do i
    ... = a[i - 1]
  end do
end if
```

The second loop can be parallelized, and only it will be executed if the input set is parallelizable.

3.3.3 Harmless Flow

Counting only the number of pairs of references for which the memory disambiguation domain fails does not give one a full picture. Some failure cases may not interfere with

parallelization at all. There are cases where, for example, a particular subroutine is never executed. An unexecuted subroutine has no loop-carried true data-flow dependences. Our compiler may assume that two references in such a subroutine have a loop-carried true data-flow dependence and may not parallelize their enclosing loops. Nonetheless, since these loops are never executed, it does not matter. Our compiler has not inhibited parallelization.

Similarly, there are loops that dynamically have only one iteration. A loop with only one iteration cannot have a loop-carried true data-flow dependence. Again our compiler may not be able parallelize these loops, but parallelizing them would not improve performance.

3.4 Affine Failures

Next we discuss in detail the affine failures. Table 11 divides these into the cases we have seen. The cases labeled as *other* are small problems that could be fixed without significantly changing our model. We merely include them for completeness.

Table 11: Categorization of affine failures

		Affine Failures				
Program	#Lines	Indirect	Symbolic	Nonlinear	Dynamic	Other
LG	2,327	0	0	0	0	0
LW	1,237	0	0	5	0	0
SD	7,607	14	0	0	0	5
SR	3,970	9	11	0	3	0
TI	484	0	0	0	0	0
TOTAL	15,625	23	11	5	3	5

3.4.1 Indirect Array References

Our model is not able to handle indirect array references such as:

```
do i
  a[b[i]] = a[b[i]]+3
end do
```

In the general case, this loop cannot be run in parallel. If, however, b is a permutation array, the loop can be run in parallel without any modifications. All the indirect reference cases we have seen, where llpp tells us there is no dependence, appear to be permutation arrays. User assertions or language extensions would be useful to identify these permutation arrays.

3.4.2 Symbolic Analysis

In Chapter 2, we described how we extended our affine model to handle some symbolic analysis. We use a scalar prepass to discover which variables can be expressed as affine functions of the induction variables, and we treat any other variables as unbound induction variables. Our treatment of symbolic variables is less general than advocated by Haghighat [21] and by Lichnewsky and Thomasset [33]. Both groups advocate finding the convex hull of all scalar constraints interprocedurally and annotating the array references with these constraints. They then use these annotations to help prove independence. This approach enables them to solve more systems exactly than our approach. Consider:

```
read(n)
if n > 10 then
  do i = 1 to 10
    a[i+n] = a[i]+3
  end do
end if
```

For this example, they annotate the two references with the fact that $n > 10$. This allows them to prove that the two references are not dependent. The more general approach can be expensive at compile time. It is necessary to track the relationship of

all scalar variables in a program. In addition, the integer programming problem being solved becomes more complex, containing more variables and more constraints.

Our approach fails due to lack of symbolic information for 11 pairs, all in one program. All 11 pairs look like the following:

```
program
  if (x) then
    n = max - 1
  else
    n = max
  end if
  call fred
end

subroutine fred
  do i
    if (x) then
      a[max] = ... (annotate with n = max - 1)
    end if
    ... = a[n]
  end do
end
```

The two references are independent. If x is true then $n = max - 1$ and n cannot equal max. If x is not true, the write never happens. To know that the two references are independent, we need to annotate the write with $n = max - 1$, but to do this, we need to know in subroutine fred that $n = max$ or $n = max - 1$ depending on the value of x. This constraint is not convex. Therefore, even the more general symbolic techniques mentioned would not be able to solve this case. Thus our inexpensive approach is able to solve all the cases we have seen that could be solved by the more general system.

3.4.3 Nonlinear Expressions

Programmers sometimes hand linearize multidimensional arrays. They convert code like the following:

```
do i
  do j
    a[j + c][i] = a[j + c2][i]+3
  end do
end do
```

into code that uses one-dimensional arrays like:

```
do i
  do j
    a[n * i + j + c] = a[n * i + j + c2]+3
  end do
end do
```

While the first example can be handled easily by our system, the second cannot. The use of the nonlinear term, ni, violates our model. We would like to be able to delinearize the arrays, converting them back into the multidimensional form.

The two references in the nonlinearized code are dependent iff

$$i_w = i_r \text{ and } j_w + c = j_r + c_2 \text{ subject to the bounds} \quad (1)$$

The two references in the linearized version are dependent iff

$$ni_w + j_w + c = ni_r + j_r + c_2 \text{ subject to the bounds} \quad (2)$$

It is clear that a solution to 1 implies a solution to 2. Now assume there is a solution to 2. Equation 2 can be rewritten as:

$$ni_w - ni_r = j_r - j_w + c_2 - c$$

The left hand side is a multiple of n. Let us assume that the magnitude of the right hand side is less than n. Then, the left hand side must equal 0 and:

$$ni_w = ni_r \text{ and } j_w + c = j_r + c_2$$

and there is a solution to 1. Thus, as long as $abs(j_w - j_r + c - c_2) < n$ for all j_w, j_r, we can safely delinearize the two references.

Maslov has independently developed a more formal algorithm for delinearization [34]. His algorithm is more detailed and little bit more general than ours.

Other examples, which at first do not appear to be linearized, are equivalent to our linearized example. For example:

```
count = 0
do i = 0 to n - 1
  do j = 0 to n - 1
    a[count] = a[count]+1
    count = count + 1
  end do
end do
```

Using induction variable substitution, we can make the substitution $count = ni + j$. After the substitution, this example is exactly the same as the explicitly linearized case, and our delinearizer will be able to parallelize it.

Linearized arrays do not appear to occur very frequently in our study. We believe that if we also measured output dependences, we would see more such arrays. While examples like the one above may not be that common, others like the following may be.

```
count = 0
do i
  do j
    a[count] = 0
    count = count + 1
  end do
end do
```

There is no read and therefore no true data-flow dependence. Nonetheless, if we do not delinearize this array, we will have to assume that there is an output dependence between the write and itself. We would like to be able to recognize this case.

Delinearization is also important when performing interprocedural data dependence analysis. In Chapter 5, we will show that without delinearization, we will not be able to accurately summarize key subroutines in one of the PERFECT Club programs.

3.4.4 Dynamic

As with the memory disambiguation model, the affine model can also fail due to dynamic dependences. There are array pairs that are dependent for some of the, but not all of the, values of the input data. For example:

```
read(n)
do i = 10 to 20
  a[i] = a[n]+3
end do
```

It may be true that n never equals i for the inputs we see in practice. As with dynamic memory disambiguation failures, versioning can be used to solve this problem.

3.5 Chapter Conclusions

We have presented a methodology for evaluating the effectiveness of data dependence analyzers. Using a compiler that can solve the affine memory disambiguation problem exactly and a dynamic tracer that can solve the data dependence problem exactly, we are able to show that affine memory disambiguators are not powerful enough to find parallelism in practice.

Most of the failure cases are due to the memory disambiguation approximation. Analyzing these cases shows us that array privatization is a vital technique for parallelizing loops. We also show that our simple, efficient symbolic scheme works as well as the more complicated, general schemes for the benchmarks we have used. Finally, we advocate delinearizing linearized arrays.

These results show that current data dependence analyzers are not powerful enough, but, we feel that using the techniques we mention, compilers will be able to solve the large majority of intraprocedural cases seen in practice.

Chapter 4

Data-flow Dependence Analysis

Traditional data dependence analysis has concentrated on solving the memory disambiguation problem. Memory disambiguation is an extension of alias analysis to individual array elements. Two references are dependent if they refer to overlapping locations. For a long time, scalar optimizers have gone beyond this type of analysis and utilized data-flow analysis to calculate how values flow through a program. Given a use of x, data-flow analyzers do not care about all the definitions to x, they care about the reaching definitions of x, those definitions that produce a value for x [2]. Similarly, we showed in the last chapter that memory disambiguation is not sufficient for parallelization. We need to extend our analysis to perform data-flow analysis on individual array elements, what we call data-flow dependence analysis. Given a use of an array element, we do not want to compute all the definitions to the same array element location, only the reaching definitions.

Other researchers have come to similar conclusions. Eigenmann et al. [13] discuss the techniques used to hand-parallelize four of the PERFECT Club programs. They show that commercial parallelizers are not powerful enough to parallelize these real programs. In particular, they find array privatization to be useful in all of the programs. In two of the four programs, 98% of the dynamic execution time of the program is spent in loops that require array privatization. That is to say that without privatization 98% of the dynamic time of these programs could not be parallelized. To privatize arrays, memory disambiguation is not sufficient; data-flow dependence analysis is required. Singh and

Hennessy perform a similar study on two of the PERFECT Club benchmarks and also find privatization to be important [47].

In this chapter, we introduce the concept of data-flow dependence vectors. Data-flow dependence vectors provide us with a clean and simple representation to extend standard direction and distance vectors. We will describe a set of algorithms developed by Feautrier, which we can adapt to compute data-flow dependence vectors [16][15][17]. Unfortunately, we believe that Feautrier's algorithms are too expensive to be used in general purpose compilers. We introduce new and efficient algorithms for the same problem that are as accurate as Feautrier's in the large majority of cases seen in practice. In order to utilize the data-flow dependence information to parallelize loops, we must privatize arrays. We therefore close this chapter by presenting an algorithm for array privatization. A description of the algorithm is available in [36]. A description of how data-flow dependence analysis fits in with more standard data dependence information can be found in [35].

4.1 Data-flow Dependence Vectors

Distance and direction vectors provide us with a representation to summarize a set of dependent iteration pairs. For example, two references are dependent with distance vector $\vec{d}$ if there exists a dependent iteration pair $(\vec{i}, \vec{i'})$ such that $\vec{i'} - \vec{i} = \vec{d}$.

With data-flow dependence analysis, we are no longer interested in all the data dependence iteration pairs. We are only interested in true data-flow dependences. We can analogously define data-flow dependence pairs. Two references are dependent with the data-flow dependence iteration pair $(\vec{i}_w, \vec{i}_r)$ if the value read by the read in iteration $\vec{i}_r$ was written by the write in iteration $\vec{i}_w$. Using data-flow dependence pairs, we can define data-flow distance and data-flow direction vectors using the same summaries that we use for standard distance and direction vectors. For example, two references are data-flow dependent with distance $\vec{d}$ if there exists a data-flow dependence pair $(\vec{i}_w, \vec{i}_r)$ such that $\vec{i}_r - \vec{i}_w = \vec{d}$.

Consider the following example:

```
do i = 0 to 1
  do j = 0 to 1
    a[i] = a[i − 1]
  end do
end do
```

For these two references to be dependent, $i_w = i_r - 1$ while j_w and j_r can take on any value within their bounds. Thus, the two references are dependent with the dependent iteration pairs $((i_w, j_w), (i_r, j_r)) \in \{((0, 0), (1, 0)), ((0, 0), (1, 1)), ((0, 1), (1, 0)), ((0, 1), (1, 1))\}$ giving the three distance vectors (1, 0), (1, 1) and (1, -1).

Looking at the data-flow dependences, i_w must still equal $i_r - 1$, but the only data-flow dependences occur when $j_w = 1$. Thus, the two references are dependent with the data-flow dependent iteration pairs $((i_w, j_w), (i_r, j_r)) \in \{((0, 1), (1, 0)), ((0, 1), (1, 1))\}$ giving the two data-flow distance vectors (1,-1) and (1,0).

Data-flow dependence vectors allow us to cleanly extend distance and direction vectors to deal with data-flow dependences. As we shall show at the end of the chapter, they will allow us to integrate privatization with parallelization.

4.2 Last Write Trees

In this section, we introduce the concept of the Last Write Tree, LWT. LWTs give us a way to represent all the data-flow dependence pairs. Given an LWT for two references, we can easily compute the data-flow dependence vectors for the two references.

Given a read iteration, $\vec{i}_r$, there can be at most one dynamic write iteration, $\vec{i}_w$, that writes the value consumed by the read. This value-producing write is the last write before the read to the same location. We can therefore define a function that given a write statement and a read statement, maps each read iteration, $\vec{i}_r$, into an expression for the last write iteration, $\vec{i}_w$, before the read to the same location.

We first motivate this representation with a simple example:

```
do i = 11 to 20
  do j = 11 to 20
    a[j] = ...
    ... = a[j - 1]
  end do
end do
```

It is easy to see that the data read in the first iteration of the innermost loops (j=11) are not defined within the loop nest. For all other iterations, the data read are written in the very preceding iteration. This information can be organized in a binary tree as shown in Figure 3. The tree represents a function that maps an instance of the read operation to the last write instance to the same location. The read/write instances are denoted by the values of their loop indices, $\vec{i}_r$ and $\vec{i}_w$, respectively. The domain of the function is the set of $\vec{i}_r$ that satisfies the constraints imposed by the loop bounds. Each internal node in the tree contains a further constraint on the value of $\vec{i}_r$. It partitions the domain of read instances into those satisfying the constraint, represented by its right descendants, and those not satisfying the constraint, represented by its left descendants. In other words, the constraints of a node's ancestors make up the node's *context*. The *iteration set* of a node is defined as the set of $\vec{i}_r$ that satisfies the node's context. Each leaf node has a solution that expresses the last write instance $\vec{i}_w$ in terms of $\vec{i}_r$, for all those $\vec{i}_r$ in its iteration set. If there is no preceding dependent write for a particular iteration set, the solution of the corresponding leaf is denoted by $\perp$.

We can formally define the LWT function as follows:

Definition 4.2.1 *Given two loop iteration vectors, $\vec{i}$ and $\vec{i}'$, corresponding to two statements nested in n common loops and an arbitrary number of non-common loops. Vector $\vec{i}$ is* **lexicographically less** *than vector $\vec{i}'$, written $\vec{i} \prec \vec{i}'$, iff*

$$\exists k \leq n, i_1 = i'_1, \ldots, i_{k-1} = i'_{k-1}, i_k < i'_k$$

Similarly, $\vec{i} \preceq \vec{i}'$ if $\vec{i} \prec \vec{i}'$ or $\vec{i}_1 = \vec{i}'_1, \ldots, \vec{i}_n = \vec{i}'_n$.

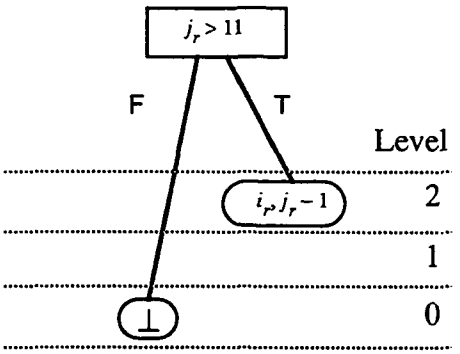


Figure 3: Example of an LWT

Definition 4.2.2 *Let $\vec{i}_w, \vec{i}'_w$ be iterations corresponding to a write statement in an m-deep loop nest, $\vec{i}_r$ be an iteration corresponding to a read statement in an n-deep loop nest, and $\vec{s}$ be a p-element symbolic constant vector in the loop. We define*

$\vec{u}_w = (i_{w1} \ldots i_{wm}, s_1 \ldots s_p)$, $\vec{u}'_w = (i'_{w1} \ldots i'_{wm}, s_1 \ldots s_p)$, $\vec{u}_r = (i_{r1} \ldots i_{rn}, s_1 \ldots s_p)$.

Let F_w, F_r be affine read and write access functions in the loop, and B_w, B_r be affine read and write loop bound constraints.

If F_w appears after F_r lexically in the program, then $LWT_{F_w,F_r,B_w,B_r}(\vec{u}_r) = \vec{i}_w$ if and only if

$$B_w\vec{u}_w \geq \vec{0}, F_w\vec{u}_w = F_r\vec{u}_r, \vec{i}_w \prec \vec{i}_r$$

and

$$\nexists \vec{i}'_w (\neq \vec{i}_w) \text{ such that } B_w\vec{u}'_w \geq \vec{0}, F_w\vec{u}'_w = F_r\vec{u}_r, \vec{i}_w \prec \vec{i}'_w \prec \vec{i}_r$$

If instead, the write statement appears lexically above the read, we must replace the condition $\vec{i}_w \prec \vec{i}_r$ in the definition with the condition $\vec{i}_w \preceq \vec{i}_r$.

4.3 Data-flow Dependence Vectors from LWTs

We can use an LWT to easily calculate the data-flow dependence vectors. We visit every leaf in the tree that is not labeled $\perp$. The context and the solution of the leaf impose a set of linear constraints on the iteration variables that can be expressed in matrix form as $A\vec{i} \geq 0$ where $\vec{i}$ is the combined vector $(\vec{i}_w, \vec{i}_r, \vec{s})$. This is exactly the same formulation as for the affine memory disambiguation problem. We can solve for the data-flow vectors in the same way that dependence analysis solves for direction vectors using a variation of Burke and Cytron's method [10]. We then return the set union of all the data-flow dependence vectors calculated at each leaf.

The concept of dependence levels can be used to optimize our algorithm for calculating data-flow dependence vectors. We define the level of a dependence as follows [3].

Definition 4.3.1 *A dependence vector or a data-flow dependence vector* $\vec{d} = (d_1, \ldots, d_m)$ *is said to be carried at level* k *if* $(d_1, \ldots, d_{k-1}) = (0, \ldots, 0)$ *and* d_k *is* $+$.

Definition 4.3.2 *The level of a dependence iteration pair is the level of the data-flow dependence vector associated with that pair.*

As described below, we calculate LWTs one dependence level at a time. All the data-flow dependence pairs described by a leaf are at the same level. Burke and Cytron's method works by enumerating and verifying possible direction vectors. If the dependence level of a leaf is k, any data-flow dependence vector with a direction other than 0 in any of its first $k-1$ dimensions is infeasible. We can immediately prune away any such vector. Given an LWT, we can calculate data-flow dependence vectors this way as efficiently as calculating direction vectors.

Different algorithms for constructing LWTs might not guarantee that each leaf only describes single-level dependences, but it is always possible to split a multi-level leaf into several nodes with only single-level leaves.

To simplify our presentation, we incorporate dependence levels directly into our trees. We draw our LWT so that each leaf's height is equal to its dependence level. Any leaf that is $\perp$ is considered to be a level 0 leaf. Any leaf describing a dependence such that $\vec{i}_w = \vec{i}_r$ is considered to be a level $n+1$ leaf.

4.4 Calculating LWTs

Feautrier has developed a set of algorithms that can be used to calculate LWTs in the domain of loop nests that contain no IF statements, no subroutine calls and no non-affine terms [16][15]. His first algorithm, Parametric Integer Programming, is based on modifying simplex for the integer case with symbolic parameters. This algorithm finds, for all array locations, the last write where the write iteration, $\vec{i}_w$, is subject to a set of linear integer constraints, $A\vec{i}_w \geq \vec{b}$, where the vector $\vec{b}$ is a symbolic constant vector.

Finding the last write before a read access cannot be directly expressed in the same framework. The constraint that the write comes before the read is a lexicographic constraint that cannot be represented in the form of $A\vec{i}_w \geq \vec{b}$. The parametric integer programming algorithm is able to find the lexicographically last write subject to a set of linear constraints but not the lexicographically last write subject to a lexicographic constraint. Finding the last write at a given dependence level, however, can be expressed in this framework. Feautrier's second algorithm therefore solves for the LWT on a level by level basis. If, for a certain value of $\vec{i}_r$, a dependence at level k exists, then there cannot be a later dependent write with a lower dependence level. Assume first that the read statement comes lexically before the write statement in the program text. Feautrier first solves for the last write within the innermost loop level, say, n. This generates a sub-tree. If a particular leaf of the subtree has a last write solution, he has found the last write before the read for the read iteration set. If a leaf is $\perp$, it represents a read iteration set with no level n dependences. For each such leaf, he then searches for the last write at level $n-1$, replacing the leaf with the newly generated level $n-1$ subtree. Each recursive call inherits the context of its leaf. This insures that all solutions with a level k dependence represent read iterations sets that do not have higher level dependences. This procedure generates the full LWT.

If the write statement comes lexically before the read, it is possible that there is a true data-flow dependence when $\vec{i}_w = \vec{i}_r$. We can consider this to be a level $n+1$ dependence. In this case, the algorithm is slightly modified to start the recursion at level $n+1$ instead of at level n.

Using Feautrier's algorithm, for each array pair, we are required to solve exponentially many parametric integer programming problems for each dependence level. Each parametric integer programming problem itself requires solving a large number of integer programming problems. As an example, Feautrier timed his algorithm on the following test input case [17].

```
do i = 1 to l
  do j = 2 to n - 1
    do k = 2 to n - 1
      a[j][k]=0.25*(a[j][k-1]+a[j][k+1]+a[j-1][k]+a[j+1][k])
    end do
  end do
end do
```

Feautrier found that his algorithm takes 1.7 seconds to solve this input on a low end SPARC [17]. We have also implemented Feautrier's algorithms; we find them complex and obtain similar timing figures. We believe that the algorithm is much too inefficient for use on real programs.

Our algorithm described below is designed to exploit the inherent simplicity in real code. It can find exact data-flow information for the vast majority of programs very efficiently. On Feautrier's example that took 1.7 seconds, our algorithm takes on a similar machine, a DECstation 3100, about 0.10 seconds to both calculate the LWT and to compute the data-flow dependence vectors. The efficiency of our algorithm makes it feasible to use array data-flow analysis in program optimizations.

In the following, we first concentrate on finding the array data-flow information between a pair of read and write operations in a loop nest with affine loop bounds. To explain the algorithm, we describe our observations on two degenerate cases: when a write access touches the same locations in all instances and when a write access touches all different locations in different instances. We then show how we can use these ideas as building blocks to develop an algorithm applicable to many of the cases found in practice. We will back up the claim of its generality by some empirical evidence. Finally,

we describe how we handle multiple write operations and discuss ways to expand the domain in which this algorithm applies.

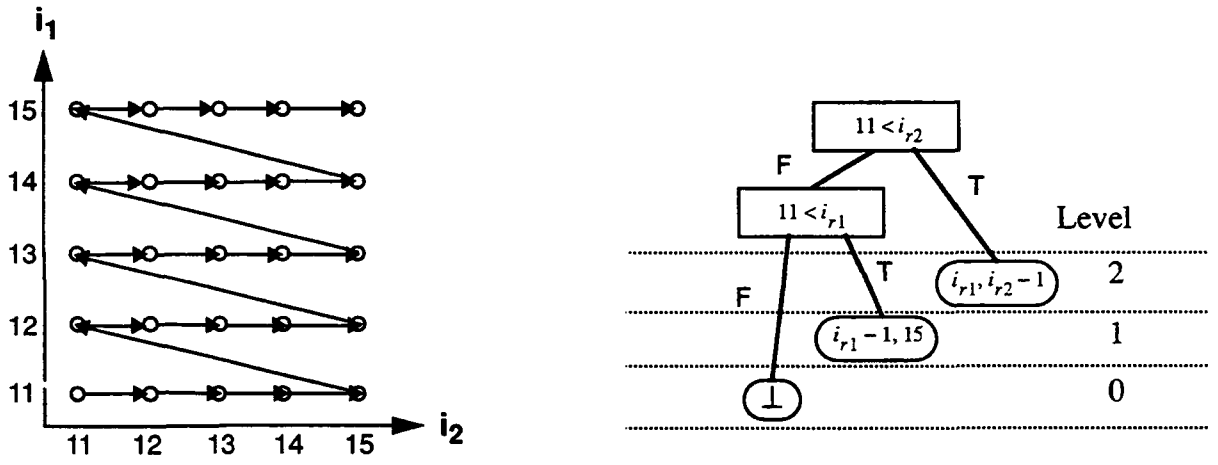
4.4.1 Loop Independent References

We say that a statement is loop-independent if it refers to the same location on every iteration. For example:

```
do i1 = 11 to 15
  do i2 = 11 to 15
    ... = a
    a = ...
  end do
end do
```

every write iteration writes to the same location being read. To find the last dependent write before the read, we merely need to compute the last write before the read. In the first iteration, we are reading an uninitialized value. In every other iteration, the last write before the read is one iteration before the read. In Figure 4.a, we show a graphical representation of the iteration space ordering for the example. From the figure we can see that if i_{r2} is not at its minimum value, 11, then one iteration before the read is (i_{r1}, $i_{r2} - 1$). Otherwise, one iteration before the read is ($i_{r1} - 1$,15). We show the corresponding LWT in Figure 4.b.

In general, assume we have a loop-independent read and write, accessing the same location and perfectly nested in n common loops. If the write statement comes lexically before the read, then our solution is $\vec{i}_w = \vec{i}_r$. If the read statement comes lexically before the write statement, then we create a tree similar to the one in the example using a simple, recursive procedure. We start our recursion with the innermost loop, corresponding to loop index i_n. At any level k in the recursion, we create a subtree with the root "if $i_{r_k} > LB_k$" were LB_k is the expression for the lower bound of i_{r_k} and where $\vec{i}_{r_k}$ refers to the $k'th$ component of $\vec{i}_r$. If we are working on the outermost loop, $k = 1$, we set



a. b.

Figure 4: LWT for loop-independent references

the false child of the subtree root to $\perp$. Otherwise, we set the false child to the result of applying the recursive step to level $k-1$. We set the true child of the subtree root to be the solution node $i_{w_{1\ldots k-1}} = i_{r_{1\ldots k-1}}, i_{w_k} = i_{r_k} - 1, i_{w_{k+1\ldots n}} = UB_{k+1\ldots n}$ where each UB_l is the upper bound for i_{w_l}. If an upper bound is a constant, we simply set i_{w_l} to its value. Otherwise, if the loop is trapezoidal, we substitute in the values of the other components of $\vec{i}_w$ to find its value.

If the loops are not perfectly nested, the write statement may be nested in m loops where only the outer n loops are common. The values of the extra inner loop indices do not affect the relative ordering between the write and read instance. Therefore for every non-$\perp$ solution, we set $i_{w_{n+1}}, \ldots, i_{w_m}$ to be as large as possible, their upper bounds.

In the above, we have implicitly assumed that our loops are not degenerate. We say that a loop is *degenerate* if it is possible that no iterations of the loop will be executed. There are two types of degeneracies: full and partial. In a full degeneracy the loop will either always have iterations or never have iterations depending on the values of the symbolic constants. For example:

```
do i = 1 to 10
  do j = 1 to m
  ... = a
  end do
  do j = 1 to n
  a = ...
  end do
end do
```

If $n \geq 1$, the write loop will always have at least one iteration and will therefore never be degenerate. If $n < 1$, the loop will never execute and will therefore always be degenerate. If the write loop is degenerate, every read will read an uninitialized value, and the correct LWT will be $\perp$.

Ancourt has developed an algorithm for loops with affine loop bounds based on Fourier-Motzkin elimination that can be used to detect possibly degenerate loops [5]. For full degeneracies, we can use the algorithm to derive the conditions under which a degeneracy will not occur, i.e. $n \geq 1$ in the above example. We merely insert these conditions at the top of our LWT. If a loop is degenerate, there is no write before the read and the solution is $\perp$. Otherwise, we proceed as before.

In a partial degeneracy, a loop may be degenerate in some invocations but not in others. For example:

```
do i1 = n to 15
  do i2 = 12 to i1
    ... = a
    a = ...
  end do
end do
```

The inner loop is degenerate whenever $i_1 < 12$. The algorithm we described above does not work in the presence of partial degeneracies. The boundaries of the iteration

space are no longer equal to the loop bounds. Assume for example that $n < 12$. The first executed iteration of the loop occurs when $i_1 = 12$ and $i_2 = 12$. Therefore the last write before $i_{r_1} = 12, i_{r_2} = 12$ should be $\perp$. Yet, since $i_{r_1} > n$, the LWT constructed by our algorithm would find a solution with $i_{w_1} = 11, i_{w_2} = UB_2 = i_{w_1} = 11$.

We can use Ancourt's algorithm to rewrite the loop bounds to eliminate the partial degeneracy. For the example, we can eliminate the degeneracy by replacing the lower bound of i_1 with the value $\max(n, 12)$. We must modify our LWTs to allow min's and max's in our constraints and solutions. If this is not desirable, min's and max's can always be expanded out. For example, $i < \max(n, 12)$ can be expanded into the two nodes $i < n$ and $i < 12$. After eliminating partial degeneracies in this manner, our original algorithm works correctly.

Partial degeneracies may lead to a serious complication. If the read and write statement are not perfectly nested, after eliminating the partial degeneracies, the bounds for the write statement may be different from the bounds for the read statement, even in the common loops. For example:

```
do i1 = n to 15
  ... = a
  do i2 = 12 to i1
    a = ...
  end do
end do
```

Ancourt's algorithm will change the lower bound of the index for the outer loop for the write to $i_{w_1} = \max(n, 12)$, but the lower bound of the index for the read loop, i_{r_1}, does not change since the read is never partially degenerate in the example.

Different bounds for the write and read greatly complicate the LWT. Our algorithm implicitly assumed that we can always set a particular write index, $i_{w_x} = i_{r_x}$. This is no longer possible if i_{r_x} is within the read loop bounds but not the write loop bounds. Similarly, the fact that $i_{r_x} > LB_x$ no longer guarantees that there exists an i_{w_x} such that $i_{w_x} = i_{r_x} - 1$. Therefore for the cases where Ancourt's algorithm sets the write bounds to

be different from the read bounds, we believe that Feautrier's algorithm should be used. As we shall show, this type of degeneracy is extremely rare.

4.4.2 Writes That Do Not Self-Interfere

We now consider the opposite scenario where a write operation accesses a different location in each iteration. We say that such a write does not "self-interfere". For example:

```
do i = 11 to 20
  a[i] =
end do
do i = 11 to 20
  a[3] =
end do
```

In the first loop, we are writing to different locations in each iteration so the write statement does not self-interfere, but in the second loop, we are writing to the same location, a[3], in each iteration so the write statement does self-interfere.

A write that never self-interferes will write at most one value into any location. Since each read iteration is associated with only one location, given a read iteration, there can only be at most one dependent write instance before the read instance. Thus, computing all the dependent writes before the read is equivalent to computing the last dependent write before the read, and we can easily modify our affine memory disambiguation system to generate the LWT.

For there to be a dependence, the read and write must refer to the same location. Using our array reference functions, for every dependence we know that $F_w\vec{i}_w = F_r\vec{i}_r$. To simplify the presentation, we first assume that we are interested in real solutions and that F_w is invertible. Then, we know that $\vec{i}_w = F_w^{-1}F_r\vec{i}_r$.

For every read iteration, $\vec{i}_r$, satisfying the bounds $B_r\vec{i}_r \geq 0$, the dependent write $\vec{i}_w = F_w^{-1}F_r\vec{i}_r$. Not all these writes, though, are within the write bounds $B_w\vec{i}_w \geq 0$, and not all these writes are before the read. The LWT template in Figure 5 ensures that these

constraints are met for the case where the read statement is lexically before the write statement.

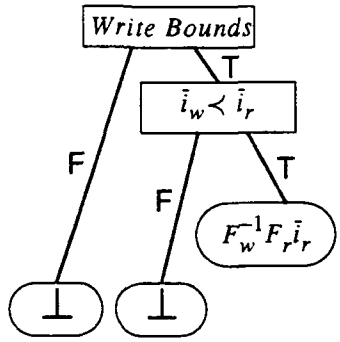


Figure 5: Template LWT for non-self-interfering write

Assume that the write statement is nested in m loops, n of which are common to the read. After expanding out the lexicographical ordering constraint and replacing in the conditional each occurrence of $\vec{i}_w$ with its value, $\vec{i}_w = F_w^{-1}F_r\vec{i}_r$, we get the LWT in Figure 6. Note that our lexicographic ordering constraint is expanded into two internal nodes for each of the n *common* loop indices, but our solution $\vec{i}_w = F_w^{-1}F_r\vec{i}_r$ is a solution for all the write loop indices $i_{w_1}, \ldots, i_{w_m}$. Note also that if the write statement comes lexically before the read, the true child of the lowest internal node, $(F_w^{-1}F_r\vec{i}_r)_n \geq \vec{i}_{r_n}$, will be $F_w^{-1}F_r\vec{i}_r$ rather than $\perp$.

Some nodes of the tree may be inconsistent. If, for example, there is no dependence such that $i_{w_n} \geq i_{r_n}$, the last if clause is always false. We can prune away this node, replacing the subtree rooted at $i_{w_n} \geq i_{r_n}$ with its false child. We can find the inconsistent nodes by testing each of the n nodes for consistency. Checking all the nodes of the tree for consistency is similar to testing for a memory disambiguation dependence once at each dependence level, a subset of standard memory disambiguation.

In the general case, we are interested in integral solutions instead of real ones, and the matrix F_w is not always invertible. The Smith normal form [44] can be used to solve $F_w\vec{i}_w = F_r\vec{i}_r$ in the general case. In a process similar to the extended GCD Test [8], which is similar to an integral version of LU decomposition, we can find matrices P, Q and S such that $PF_wQ = S$, P and Q are unimodular matrices and S is of

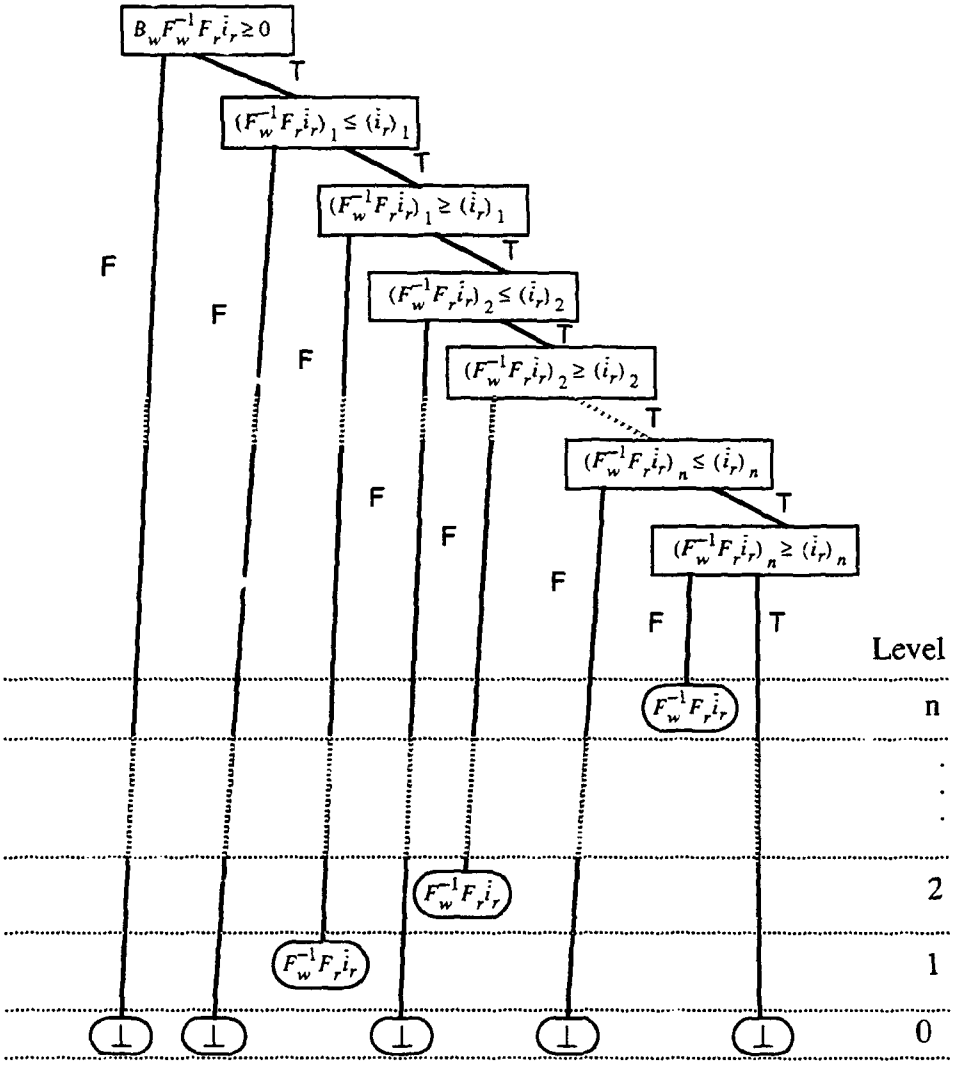


Figure 6: LWT for non-self-interfering write

the form $\begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$ where D is a positive diagonal matrix. The initial system has an integer solution if and only if there exists an integer vector $\vec{t}$ such that $S\vec{t} = PF_r\vec{i}_r$. Since D is a diagonal matrix, the constraint $S\vec{t} = PF_r\vec{i}_r$ is equivalent to the set of constraints $D_{x,x}t_x = (PF_r\vec{i}_r)_x$. An integral t_x satisfying this constraint exists if and only if $(PF_r\vec{i}_r)_x \% D_{x,x} = 0$ where % is the integer modulo operation.

If there exists an integral solution, then the solution is given by $\vec{i}_w = Q\vec{t}$. If there are no zero diagonal elements in S ($S = D$), we can solve for $\vec{t}$ in terms of $\vec{i}_r$. D^{-1} is simply the diagonal matrix such that $(D^{-1})_{i,i} = 1/D_{i,i}$ and $\vec{t} = D^{-1}PF_r\vec{i}_r$.

To create the LWT, we merely add to our tree the nodes $(PF_r\vec{i}_r)_x \% D_{x,x} = 0$. If these conditions are not met, there is no dependent write, otherwise we create a tree as before, replacing $\vec{i}_w = F_w^{-1}F_r\vec{i}_r$ with $\vec{i}_w = QD^{-1}PF_r\vec{i}_r$.

Figure 7 shows the resultant tree.

If S has a 0 in the diagonal, it is not possible to eliminate all the t variables. That is, there are multiple values of $\vec{i}_w$ for each $\vec{i}_r$ that satisfy $F_w\vec{i}_w = F_r\vec{i}_r$. However, for the write to qualify as not self-interfering, the loop bounds must impose sufficient additional constraints on the problem to reduce the solution to at most one single value of $\vec{i}_w$ for each $\vec{i}_r$. For these rather uncommon cases, we can use a memory disambiguator to determine the unique t values [40].

Note that degenerate loops are not a problem for this case. If a particular solution for $\vec{i}_w$ is invalid due to a degeneracy, one of the bounds constraints will fail, and our solution will be $\perp$.

4.4.3 Computing LWT Efficiently

We have shown how to construct LWTs for pairs whose write statements do not self-interfere and for pairs with loop-independent references. Both of these cases, by themselves, are not very interesting. When write statements do not self-interfere, LWTs contain no more information than standard memory disambiguation. The advantage of data-flow dependence analysis is to eliminate dependences that are covered by other writes. Also, not many writes are loop-independent. We can, though, use our algorithms for these two

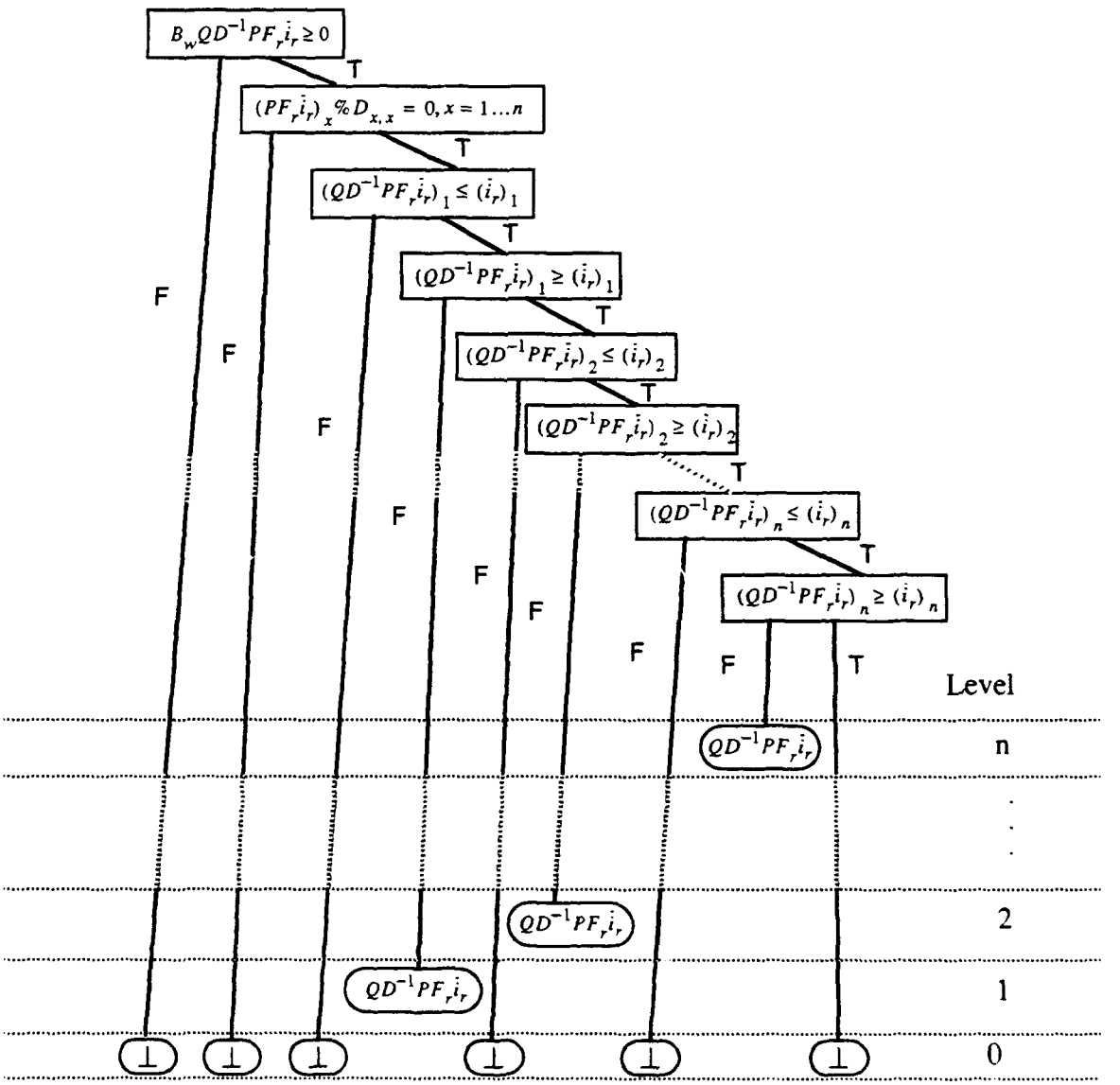


Figure 7: LWT generated using Smith Normal Form

cases as building blocks for an algorithm that is both simple and widely applicable.

Domain of Algorithm

To specify the domain of our algorithm precisely, let us first introduce the notion of *unused loop indices*. We define a loop index to be unused if it does not appear in the array reference function of the write, F_w, nor in the bounds of any used loop index. An unused index may be used by the read statement. A loop-independent write corresponds to the degenerate case when all the loop indices are unused by the write. The loop indices i and j in the code below are considered unused:

```
do i = 11 to 20
  do j = 11 to 20
    do k = 11 to 20
      a[k] = ...
      ... = a[k-1] + a[i+j]
    end do
  end do
end do
```

Our algorithm can generate the LWT exactly and efficiently if eliminating the unused loop indices reduces the write access into a non-interfering one and if Ancourt's algorithm to eliminate partial degeneracies does not introduce read bounds that are different than the write bounds in any of the removed loops. To assess the generality of our algorithm, we performed the following experiment. We inserted a pass in our SUIF compiler to examine all of the array writes in the PERFECT Club programs that have affine array index expressions and are nested in at least one loop. After eliminating the unused indices, we checked to see if each write statement self-interfered. This is equivalent to checking for an output memory disambiguation dependence from the write to itself with a non-0 direction vector component. Table 12 shows that after removing the unused indices, in 2,501 out of 2,567 writes, over 97%, the writes do not self-interfere.

Table 12: Number of writes that self-interfere after removing unused indices

Self-Interference (SI)														
	AP	CS	LG	LW	MT	NA	OC	SD	SM	SR	TF	TI	WS	TOT
SI	0	2	0	0	0	13	39	0	0	0	0	12	0	66
NSI	424	84	36	36	74	316	75	142	124	430	233	10	517	2,501

It may not be that interesting to evaluate the applicability of our algorithm on all the array writes. As we saw in Chapter 3, memory disambiguation fails for about 15% of the array pairs; the pairs which have true dependences but not true data-flow dependences. Nonetheless, memory disambiguation does succeed for most pairs. Memory disambiguation is usually sufficient because most array writes do not self-interfere, even without eliminating the loops corresponding to the unused indices. While we can handle such writes, handling them gives us no more information than standard memory disambiguation. In Table 13, we eliminate these writes. Looking only at the writes that do self-interfere, we show how many still self-interfere after removing the unused indices. Even only looking at these problematic writes, we are able to handle 566 out of 632 or about 90% of the cases.

Table 13: Number of self-interfering writes that self-interfere after removing unused indices

Self-Interference														
	AP	CS	LG	LW	MT	NA	OC	SD	SM	SR	TF	TI	WS	TOT
SI	0	2	0	0	0	13	39	0	0	0	0	12	0	66
NSI	66	19	14	24	14	49	36	16	87	82	9	8	142	566

We then did a simple check to look for partial degeneracies. Partial degeneracies can only occur in trapezoidal loops. For each trapezoidal loop, we performed a simple and efficient substitution step to try to prove that the trapezoidal loop can not be partially degenerate. For example:

do i_1 = 1 **to** n
  do i_2 = 1 **to** i_1

By substituting in its lower bound, we know that $i_1 \geq 1$ and the inner loop can never be partially degenerate.

Our substitution algorithm was able to prove that 98% of the writes in the PERFECT Club were not nested in any partially degenerate loops. Of course, our simple experiment was too restrictive. The use of symbolic variables might force our substitution algorithm to assume that a partial degeneracy exists when it does not. Also, we do not differentiate partial degeneracies in the used loop indices from partial degeneracies in the unused indices, and we do not check if partial degeneracies would cause Ancourt's algorithm to set the write bounds to be different from the read bounds. Thus our algorithm should be even more applicable.

Our Algorithm

The outline of our algorithm is as follows:

1. Apply Ancourt's algorithm to eliminate partial degeneracies.
2. Remove the loops corresponding to those loop indices that are unused by the write statement. We treat the unused loop indices remaining in the read statements as symbolic constants.
3. Check if the write in the reduced system self-interferes. If the write self-interferes, the algorithm is not applicable. Otherwise:
4. Construct the LWT for the reduced system using the non-self-interfering algorithm.
5. Construct the LWT for the original system by *restoring* the unused loop indices. We will describe this step in detail below. If Ancourt's algorithm has set any of the write bounds in the unused loops indices to be different from the corresponding read bounds, this step of our algorithm is not applicable, and we must use Feautrier's algorithm to restore the unused indices.

Note that even when our algorithm does not apply because of self-interference, our LWT is only inaccurate in that it conservatively describes more dependent writes than just the last write before the read. Thus in cases where there is reuse in some of the loops corresponding to used indices, but not all, our algorithm might still be useful in pruning some of the false data-flow dependences.

Restoring Unused Variables

Given an LWT for a reduced system with the loops corresponding to the unused write indices removed, we need to be able to construct the LWT for the original system. We assume that Ancourt's algorithm has been applied to eliminate all the partial loop degeneracies. This step may have introduced min's and max's into our loop bounds. We restrict our scope to the case when the common loop bounds of the read and write statement are equivalent. We also use Ancourt's algorithm to find the conditions that guarantee no full degeneracies. These conditions are placed at the top of our tree.

We will restore one unused index at a time starting with the innermost index. We need to show that at each stage, our algorithm generates the correct tree for the partially reduced system with all loops corresponding to the not yet restored variables removed.

Recall that the values of the non-common indices do not affect the ordering between the read and write. Therefore, for any non-$\perp$ solution node, we should set all non-common, unused write indices to their upper bound, and we should set all the non-common, used indices to their corresponding component in the solution, $QD^{-1}PF_r\vec{i}_r$. We can ignore the non-common loops for the remainder of our discussion.

We now describe the algorithm to restore the index corresponding to the $k'th$ outer loop. Assume that before this restoration, the LWT for our system is T and that our restoration process is creating a new tree, T'. Each internal node in an LWT merely partitions the domain of our LWT function. A given value of $\vec{i}_r$ and a given value of the symbolic variables map into one and only one leaf. The corresponding solution, $\vec{i}_w$, is the solution found in that leaf. Thus, when creating T', we must merely insure that each leaf contains the correct solution for all the reads in its read iteration set.

First let us consider the non-$\perp$ leaves in T. For every read in the read iteration set of a non-$\perp$ leaf in the fully reduced system there was one write to the same location

and that write preceded the read. Since by definition the unused indices do not affect the location being written, in the full system the location being written in $\vec{i}_w$ will be the same as the location read in $\vec{i}_r$ if and only if the solution for all the used indices in the full system is the same as the solution for all these indices in the reduced system. Thus to restore the unused index, i_{w_k}, to the leaf in T, we merely need to set the unused indices so that $\vec{i}_w$ is the last write before the read.

Each non-$\perp$ leaf in T has a level, l, corresponding to the minimum index such that $i_{w_l} < i_{r_l}$. If no such index exists, then $\vec{i}_w = \vec{i}_r$ in T, and we shall define $l = n + 1$ where n is the number of common loops in the original, full system. Assume first that $l < k$. Since we are restoring the innermost unused indices first, and since index l appears in T, we know that i_{r_l} corresponds to a used index. Since the restoration process cannot change the value of a used index, we know that $i_{w_l} < i_{r_l}$ in T'. Whatever value we give i_{w_k} in T', the write will come before the read. We therefore set i_{w_k} to be as large as possible, the upper bound for loop index i_k, $UB_k(i_{w_1}, \ldots, i_{w_{k-1}})$. If the upper bound is a constant, we simply set i_{w_k} to its value. Otherwise, if the loop is trapezoidal, we can use a simple substitution step to find its value. Note that we must insure that any previously restored variable retains its correct value throughout the restoration process. Since any previously restored variable, i_{w_m}, has $m > k$, we know that $m > l$ and therefore i_{w_m} has previously been set to its upper bound, which is still its correct value.

Now assume that $l > k$. Since in T, $\vec{i}_w \preceq \vec{i}_r$, we must have $i_{w_1} = i_{r_1}, \ldots, i_{w_{k-1}} = i_{r_{k-1}}$ for all the existing indices in T. If we set i_{w_k} to be larger than i_{r_k}, then the write instance will be later than the read so we set $i_{w_k} = i_{r_k}$. This is always possible since the bounds on the read are equivalent to the bounds on the writes. Now consider once again each previously restored variable, i_{w_m}. Our restoration process has not changed the level of the leaf; it is still l. Since the value of i_{w_m} was determined solely by the relative values of m and l, the value for i_{w_m} in T is still the correct value for i_{w_m} in T'.

Thus for every non-$\perp$ leaf in T, we can compute the solution in T'. Now let us consider the $\perp$ leaves. From Figure 7, we see that in the reduced system, there are three different types of $\perp$ leaves: those corresponding to out of bounds writes, those corresponding to read iterations with no write iterations referring to the same location and those corresponding to read iterations with no preceding write iterations accessing

the same location. For the $\perp$ leaves corresponding to out of bounds writes, in T' the writes will still be out of bounds. For the $\perp$ leaves corresponding to reads without any writes referring to the same location, since the unused indices do not affect the locations being written, in T' there can also be no dependent writes. Thus we leave all such $\perp$ leaves in T as $\perp$ leaves in T'.

For the $\perp$ leaves corresponding to reads with no preceding dependent writes, though, the fact that there was no preceding dependent write in T does not necessarily imply that there is no preceding dependent write in T'. For all the reads in the read iterations set of these $\perp$ leaves, in the fully reduced system each read, $\vec{i}_r$, had one dependent write, $\vec{i}_w$, but that write was after the read. We can divide these $\perp$ leaves into two sets depending on the value of that later dependent write, $\vec{i}_w$. First, assume that in the fully reduced system there exists an $l < k$ such that $i_{w_l} > i_{r_l}$. Since $l < k$, regardless of the value of i_{w_k} or any of the other previously restored indices, the write will still come after the read and these leaves remain $\perp$ leaves in T'. Now assume that no such l exists. This implies that $i_{w_1} = i_{r_1}, \ldots, i_{w_{k-1}} = i_{r_{k-1}}$ for all the existing indices in T. In this case, as long as $i_{w_k} < i_{r_k}$, there is a preceding dependent write in T'. Since our read and write bounds are identical, if i_{r_k} is greater than its lower bound $LB_k(i_{r_1}, \ldots, i_{r_{k-1}})$, such a write exists, and the latest write occurs when $i_{w_k} = i_{r_k} - 1$. If i_{r_k} is equal to its lower bound $LB_k(i_{r_1}, \ldots, i_{r_{k-1}})$, there is no $i_{w_k} < i_{r_k}$ and we must again retain the $\perp$ leaf.

Because of our non-self-interfering write algorithm, it is easy to find all the leaves where no such l exists. We merely search T for the node with the constraint $i_{w_{k'}} \geq i_{r_{k'}}$, where k' is the last used index less than k. We call this node our *dividing node*. If there is no used index, $k' < k$, our dividing node is the last dependence node, $(PF_r\vec{i}_r)_x \% D_{x,x} = 0$, the parent of the first lexicographic node in T. From the context of this node, we know that $i_{w_1} = i_{r_1}, \ldots, i_{w_{k-1}} = i_{r_{k-1}}$ for all the existing indices in T. Since every node inherits the context of its ancestors, these equality conditions are true for every $\perp$ node that is a descendent of the true edge of the dividing node. Thus, for any descendent $\perp$ node of the true edge of our dividing node, no such l exists. We therefore replace all such descendent $\perp$ nodes with the subtree whose root is the condition "if $i_{r_k} > LB_k(i_{r_1}, \ldots, i_{r_{k-1}})$" and whose false child is a $\perp$ node. The true child is the solution node with $i_{w_k} = i_{r_k} - 1$, every other unused variable set to its respective upper bound and every used variable set to its

corresponding component in the solution, $QD^{-1}PF_r\vec{i_r}$. Note that the subtree we have described is the same as the tree generated by our loop-independent algorithm for the case of a read statement followed by a write statement nested in one loop. Loop-independent references correspond to the case when all variables are unused.

Note that in the worst case, the size of the resultant LWT can be exponential. Nonetheless, in practice we have found that the size of most LWTs we construct is close to linear in the nesting level of the loops, and our algorithm appears to be very efficient in practice.

Example of Restoring Unused Variables

We shall illustrate our restoration algorithm with an example. Assume we have a write statement nested in three loops; i_{w_1}, i_{w_2}, and i_{w_3}. Assume that the read statement comes lexically before the write statement in the program text. Assume that index i_{w_2} is unused and has constant bounds $LB_2 \leq i_{w_2} \leq UB_2$. The k mentioned in our algorithm is therefore equal to 2. Assume that after removing the second loop, the write in the reduced system does not self-interfere. The LWT for the reduced system is shown in Figure 8. We label some of the internal nodes N_0, N_1 and N_2 to aid in the discussion.

Consider first the two non-$\perp$ leaves. The dependence level of the lower left leaf is 1 which is less than $k = 2$. We therefore set $i_{w_2} = UB_2$ for this leaf. The dependence level of the upper right leaf is 3 which is greater than $k = 2$. We therefore set $i_{w_2} = i_{r_2}$ for this leaf.

Now consider the $\perp$ leaves. Node N_0 represents the condition that $i_{w_1} \geq i_{r_1}$. It is the dividing node. For any $\perp$ node descended from its true edge, we know that $i_{w_1} = i_{r_1}$. There are two such $\perp$ leaves: the false child of N_1 and the true child of N_2. We replace both $\perp$ leaves with the subtree whose root is the condition "if $i_{r_2} > LB_2$", whose false child is a $\perp$ node and whose true child has the standard partial solution and $i_{w_2} = i_{r_2} - 1$.

We show the resultant LWT for the full system in Figure 9.

4.5 LWTs for Multiple Reads and Writes

We have described an algorithm to calculate an LWT for a single pair of read and write statements. For multiple read statements we can construct an LWT for each read

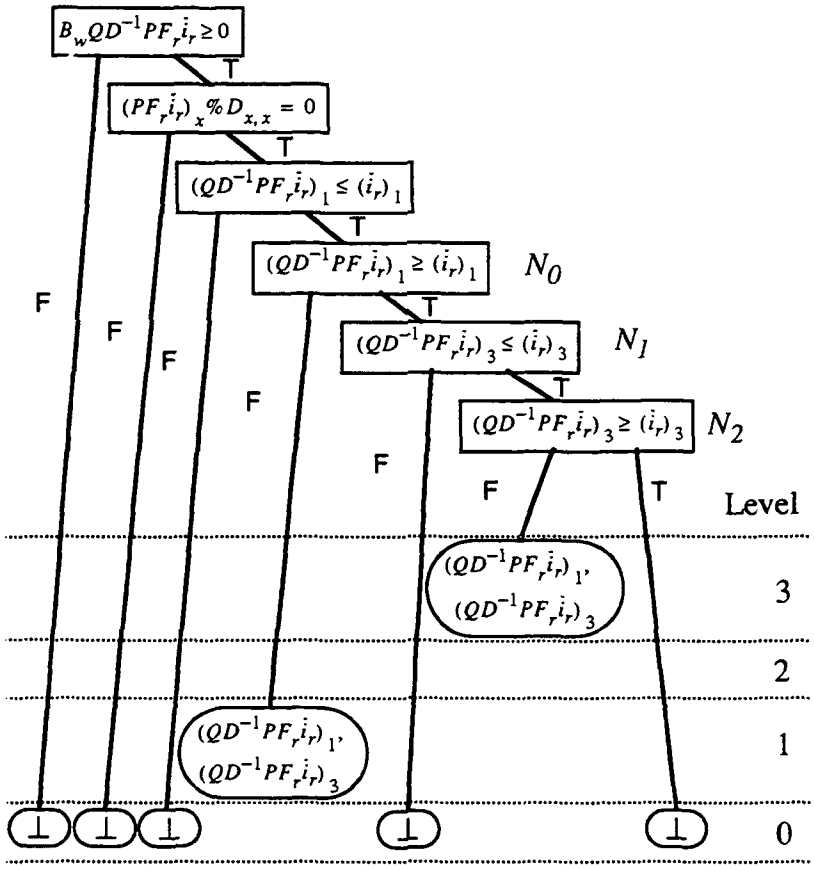


Figure 8: LWT for reduced system

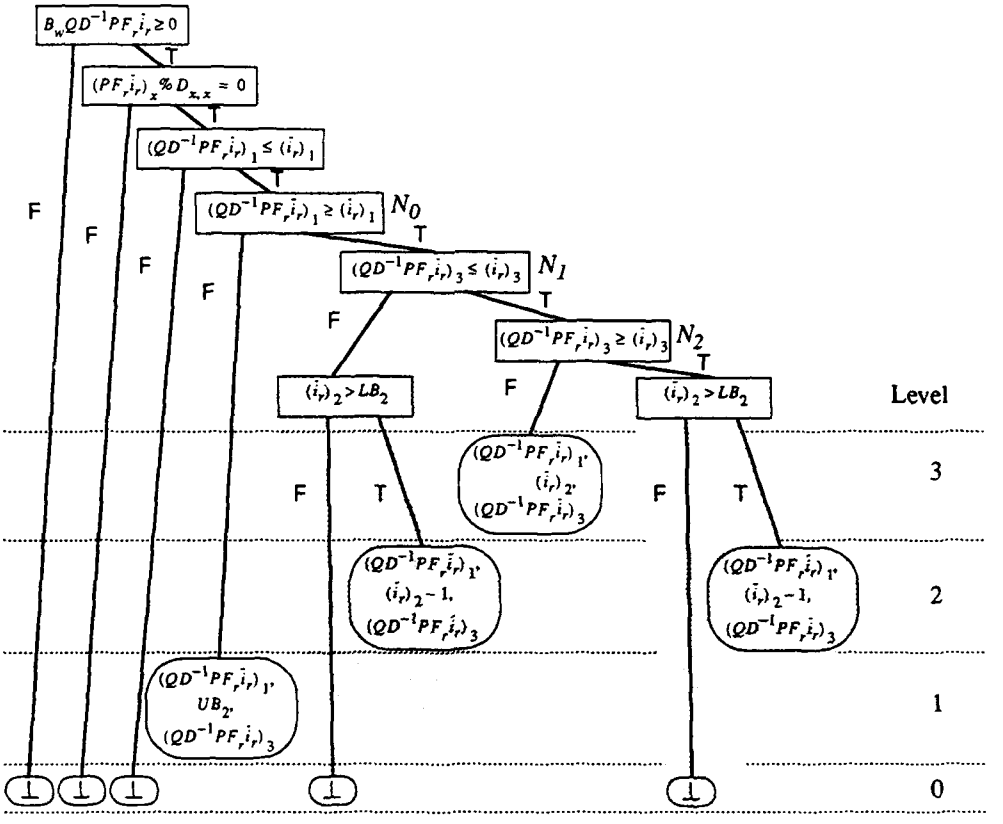


Figure 9: LWT after restoring unused variables

statement and treat them as separate problems. This is not possible when dealing with multiple writes. Take, for example, the code in Figure 10. Comparing only the first write statement to the read, we must assume that there is a true data-flow dependence because it appears that in each iteration we are reading a value that was written in the previous iteration. In fact, there is no true data-flow dependence. In each iteration, the second write statement overwrites, or covers, the location written by the first write statement in the previous iteration. The value read by the read statement was written by the second write statement in the same iteration as the read.

```
do i = 1 to 10
   a[i+1] = ...
   a[i] = ...
   ... = a[i]
end do
```

Figure 10: Multiple writes

In general, handling multiple writes requires us to intersect LWTs. Assume we have two write references and have constructed the LWTs for each reference separately. Intersecting greater than two references can always be accomplished by intersecting two references at a time. Take a leaf of the first tree at level l_1. Because of the second reference, the solution for the leaf might no longer represent the last write before the read. Any level $l_2 > l_1$ leaf from the second tree represents a later dependent write instance for any read instance common to both read iteration sets. Any leaf from the second tree with level $l_2 = l_1$ might represent a later dependent write for the common read instances if the solution for the second reference is later than the solution for the first.

The algorithm to intersect two trees, T' and T, is as follows. Assume that the write reference for T' comes lexically after the write reference for T in the program text. We visit every leaf in the first tree, T'. Assume the level of a visited leaf is l, and the solution is s'. We wish to replace s' with all the later writes from the second reference. For simplicity, assume first that the second tree, T, does not contain any level l leaves. Let T_l contain only the level $k > l$ dependences of T. Given T, we construct T_l by

replacing any level $k \leq l$ dependence in T with $\perp$. Any non-$\perp$ solution in T_l represents a dependent write that occurs later than s'. Any $\perp$ node in T_l represents reads for which there is no later write from the second reference; the last write of the two references is s'. We therefore replace s' in T' with T_l after first replacing all $\perp$ nodes in T_l with s'.

Now we consider the case when T may also contain level l leaves. Given a level l solution, s, in T, s is the latter write if and only if $s \succ s'$. Therefore, before replacing s' with the modified T_l, we first replace any level l solution in T_l with the subtree "if $(s \succ s')$ then s else s' ". Note that since s and s' are at the same dependence level, the lexicographic constraint $s \succ s'$ is equivalent to the linear constraint, $s_l > s'_l$.

In Figure 11 we show two LWTs for the examples in Figure 10. We wish to intersect the second tree, T into the first tree, T'. We first visit the $\perp$ nodes in T' and construct T_0. For any tree, T, $T_0 = T$. Therefore, we replace every $\perp$ node in T' with a copy of T. Next, we visit the level 1 leaf in T'. Since the only dependence in T is at level 2, T_1 is also equivalent to T. Thus we replace the level 1 leaf in T' with a copy of T after first replacing each $\perp$ in the copy of T with the solution from the level 1 leaf in T', $i_w = i_r - 1$.

In Figure 12, we show the resultant intersected tree. Recall that the second write statement completely covers the first write statement, and the read only sees writes from the second write statement. Therefore, the LWT for the second reference is in fact a valid LWT for the intersection of the two trees. If we eliminate redundant constraints and rearrange some other constraints, we can see that the tree in Figure 12 is equivalent to the LWT of the second write statement.

Note that our intersected tree is more complicated than either of our original trees. The complexity of our resultant tree and therefore the complexity of our algorithm grows exponentially with the number of write statements. It might be expensive in general to intersect all the LWTs corresponding to all the write statements. There are, though, two special cases that we believe are very common and that we can handle without general tree intersections. If two write statements write to identical locations in every iteration, then the latter write statement overwrites the writes of the earlier statement. The intersection of the writes is then simply the LWT for the covering write. If the loops are perfectly nested, we can simply check to see if the two write reference functions are

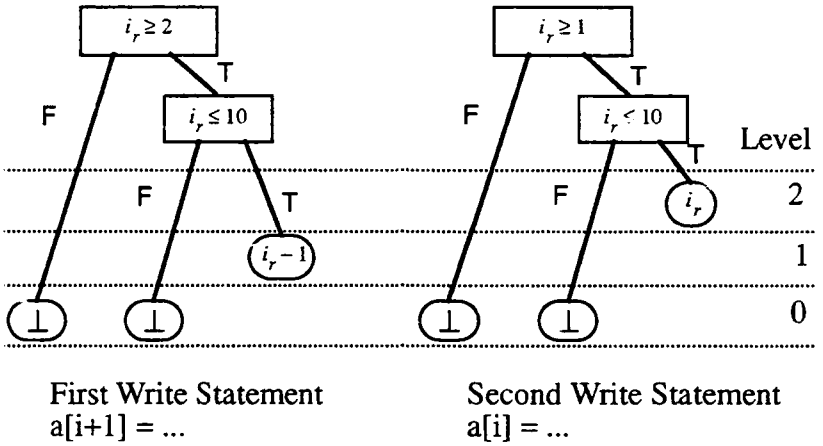


Figure 11: LWT for two writes

exactly equivalent. Otherwise, we can use a simple range based summary representation for the non-common loops to detect if the two writes write to identical locations in every iteration of the common loops.

The other simple case is when the two writes write to non-overlapping locations. This can be detected using standard affine memory disambiguation. Given a particular read iteration, $\vec{i_r}$, with its particular access location, at most one of the trees may contain a last dependent write for iteration $\vec{i_r}$. Let T' be the LWT for the first write statement, w'. Let T be the LWT for the second write statement, w. We start with T' as our prospective LWT. If $\vec{i_r}$ is dependent on a write in statement w', our tree is correct for this read instance. Otherwise, $\vec{i_r}$ will map to a $\perp$ node in T'. Since w' and w refer to non-overlapping locations, in order for $\vec{i_r}$ to be dependent on a write instance of the other write statement, w, $\vec{i_r}$ must not be dependent nor anti-dependent on any write instance of statement w'. Thus one of the bounds constraints, $B_w Q D^{-1} P F_r \vec{i_r} \geq 0$, or one of the dependence constraints,$(P F_r \vec{i_r})_x \% D_{x,x} = 0$, in T' must be false for $\vec{i_r}$. Therefore, to generate our LWT, we simply replace all the $\perp$ nodes that are false childs of a bounds

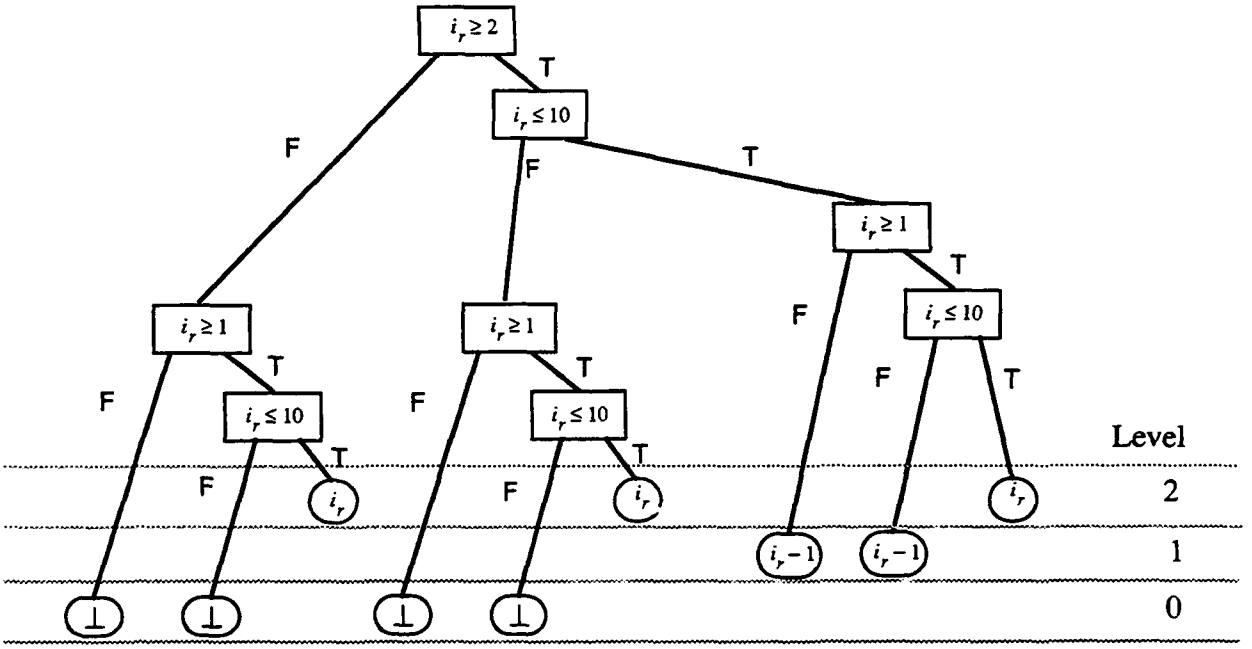


Figure 12: Intersection of two LWTs

or of a dependence constraint with a copy of T.

4.6 Array Privatization: An Optimization

We will demonstrate the necessity of more powerful data dependence representations with the array privatization example. As we have mentioned, recent research by Eigenmann et al. [13], by Singh and Hennessy [47], and by us [38] has shown that array privatization is a critical transformation in the suite of parallelizing transformations. To privatize arrays, traditional data dependence techniques are not sufficient; data-flow dependence vectors are required.

In this section we will outline a compiler algorithm for array privatization in the domain of loop nests that contain no IF statements, no subroutine calls and no non-affine terms. Our intention is to illustrate the necessity and sufficiency of data-flow dependence analysis for array privatization in this domain. A more detailed and formal description of our algorithm is found in [36].

Consider the example in Figure 13 extracted from the PERFECT Club benchmark program OCS. We will use this example throughout this section to illustrate the array privatization problem. Every iteration of the outer loop reads and writes the same elements of the array `work`. It can be shown in general that any loop, k, can be parallelized using privatization if the loop has no level k data-flow vectors. In this example, the data-flow distance vector between any two references to the work array is (0). There is no level 1 data-flow dependence, and therefore the loop can be parallelized with privatization. Using memory disambiguation, a system would detect a dependence across the iterations of the outermost loop. The outermost loop would therefore not be parallelizable.

Privatizability implies that no communication is necessary between iterations of the loop, but each processor may need to initialize its private copy and to copy back some data from the private copy to the original variable at the end of the execution. We refer to the former process *initialization* and the latter *finalization.*

In our example, every work location read is initialized in the loop. There is no need for an initialization phase. We could modify the example slightly so that some of the data being read is initialized outside of the loop. For example:

```
do j = 1 to N1 by 2
  do i = 1 to N2P
    ii = i + i
    work[ii - 1] = A[i][j]
    work[ii] = A[i][j + 1]
  do i = 1 to N2P
    A[i][j] = work[i]
    A[i][j + 1] = work[i + N2P]
```

Figure 13: Code from PERFECT Club benchmark OCS

```
work[0] = 0
do j = 1 to N1 by 2
  do i = 1 to N2P
    ii = i + i
    work[ii - 1] = A[i][j]
    work[ii] = A[i][j + 1]
  do i = 0 to N2P
    A[i][j] = work[i]
    A[i][j + 1] = work[i + N2P]
```

In the modified example, location `work[0]` is written before the start of the loop. It is still legal to privatize; the only data-flow dependence vector within the loop is still (0). Now, though, we must initialize each processor's local value of `work[0]` with the global value of `work[0]`.

To calculate the locations that need to be initialized, we use the following algorithm based on LWTs. Any array element read for which the last write before the read is not in the same iteration, must be initialized. Assume we are privatizing loop i. We must initialize all locations corresponding to level $0 \ldots i-1$ dependences. We find all the level $0 \ldots i-1$ leaves and find the union of all the array locations described by the leaves. For initialization, we can always be conservative and initialize more than necessary. Being

conservative does not limit parallelization although it does slightly increase the overhead of privatization.

For both the original example of Figure 13 and the modified version, we must copy some of the values from the local arrays back to the global after executing the loop. For these examples, the only values seen after the loop are written in the last iteration, $j = N1$. Thus, we can strip the last iteration and have the processor assigned to this iteration write to the global array rather than to its local array. It can be shown that when all the reuse is in the loops corresponding to the unused loop indices, stripping the last iteration is always sufficient for finalization.

We have implemented our algorithm in the SUIF compiler system. Our parallelizer marks all the parallelizable loops and privatizes arrays when needed. It privatizes using our efficient algorithm for calculating LWTs.

In Figure 14 we show, as an example, the output of our compiler on the code from Figure 13. Besides parallelization and privatization, the SUIF optimizer has to first perform loop normalization, constant propagation and induction variable identification on the code. The code to be run by every processor is shown. To simplify the presentation, we do not parallelize the inner loops although they can be parallelized using standard techniques.

Note that it is not easy to discover that initialization is unnecessary for this example. Each read gets part of its data from each write. Therefore, we can not simply look at pairs of references. The effects of the two writes must be combined by intersecting the two individual LWTs. Since the two writes write to non-overlapping locations, our system efficiently determines that initialization is unnecessary. As for finalization, since all the iterations of the loop write to the same locations, the final values of the array are determined only by the last iteration. Since no initialization is needed, the last processor can directly operate on the original copy of `work` and no synchronization is necessary.

4.7 Related Work

Several other researchers have worked on computing data-flow information for array elements. We have compared our work throughout this paper to Feautrier's because

```
LowBound = 0
UpBound = divfloor(N1 - 1, 2)
BlkSz = divceil(UpBound - LowBound, NumProc)
/* Body */
if (PID! = NumProc - 1) then /* not the last processor */
  do j = LowBound + PID * BlkSz to LowBound + (PID + 1) * BlkSz - 1
    do i = 1 to N2P  /* _work_priv is a local array */
      _work_priv[2*i - 1] = A[i][2*j + 1]
      _work_priv[2*i] = A[i][2*j + 2]
    end do
    do i = 1 to N2P
      A[i][2*j + 1] = _work_priv[i]
      A[i][2*j + 2] = _work_priv[i + N2P]
    end do
  end do
else /* last processor */
  do j = LowBound + PID * BlkSz to UpBound
    do i = 1 to N2P
      work[2*i - 1] = A[i][2*j + 1]
      work[2*i] = A[i][2*j + 2]
    end do
    do i = 1 to N2P
      A[i][2*j + 1] = work[i]
      A[i][2*j + 2] = work[i + N2P]
    end do
  end do
end if
```

Figure 14: Resultant privatized code

Feautrier's algorithm is exact for his domain. No one else is able to compute exact information for all inputs in his domain. As we have shown, we are as accurate as Feautrier in most cases, but we are much more efficient. Since, we know when we are exact, we can also always use Feautrier's algorithm as a backup in the rare cases when we fail. Several other researchers have worked on similar problems. Brandes [9], Ribas [42] and Pugh and Wonnacott [41] take similar approaches to us in that they extend memory disambiguation techniques to compute data-flow dependence information. Brandes's approach does not apply if dependence distances are coupled or if the loop is non-rectangular. Ribas only discusses constant-distance dependences in perfectly nested loops. Pugh's and Wonnacott's algorithm is inaccurate when their Omega Test generates disjunctions of constraints. It will have difficulties in some cases that we can handle such as when a read statement is covered by write instances from multiple dependence levels. In addition, it may have difficulties when the array reference function contains holes (i.e. a reference such as `a[2i]`).

Several other researchers have approached the problem from the scalar data-flow framework [19][20][43]. Instead of representing an array with a single bit, the set of data touched within a region/interval in the flow graph is approximated by an area descriptor. These approaches handle arbitrary control flow, which we can not, but their accuracy is limited by the precision of the summary information. As a simple example:

```
do i = 0 to n
  a[2*i] = ...
  ... = a[i]
end do
```

This loop can be parallelized in our framework with copying, a special case of array privatization. A direct summary approach would say that the write and the read sometimes refer to the same location. More information is needed for parallelization. As another example consider the PERFECT Club code fragment from Figure 13. If the summary representation is not detailed enough to describe the fact that each write only writes every other location, then parallelization would not be possible.

Li has used a similar approach in developing an algorithm specifically for the array privatization problem [30]. His algorithm handles more general control flow than ours but less general array references.

We believe that the best solution is to develop an integrated framework that combines our approach with summary based approaches. This could allow us to combine the accurate array reference analysis of our approach with the more accurate control flow analysis of summary information.

4.8 Future Work

We have given an efficient algorithm for calculating LWTs in the domain of loop nests that contain no non-affine terms, no goto statements, no subroutine calls and no conditional write statements. We have shown that our algorithm is almost always completely accurate in this domain. If desired, we can use Feautrier's algorithm as a backup. Thus, we can be completely accurate while retaining efficiency since Feautrier's algorithm would rarely have to be executed.

While our domain is the same as Feautrier's and richer than the domains of some of the other works we have mentioned, we believe that this domain must be extended in order to be useful for real programs. In Table 14, we looked at all the *do loops* in all the PERFECT Club programs and checked how many loops meet these domain restrictions. In a nested loop, we counted each loop separately since it might be possible for example to analyze and optimize the inner but not the outer loop of a doubly nested loop. We see that 1,143 out of 2,812, over 40% can not be analyzed. While we do not know the percentage of the dynamic execution time of the programs that is spent in these failure loops, we do have at least an indication that we must extend our domain restrictions.

In Table 15, we show how many of the bad loops are due to each reason: non-affine loops or non-affine array writes, gotos inside the loop, subroutine calls that write array data and conditional write statements. Note that the columns add up to more than the number of bad cases in Table 14 since a single loop can have more than one problem.

Our largest class of failures is due to the affine conditions. In Chapter 3, we showed evidence that the affine restriction is sufficient for standard memory disambiguation.

Table 14: Number of do loops that meet domain restrictions

Domain Restrictions			
Program	Good	Bad	Total
AP	195	105	300
CS	180	134	314
LG	44	115	159
LW	25	31	56
MT	58	25	83
NA	156	113	269
OC	85	76	161
SD	169	93	262
SM	56	248	304
SR	210	20	230
TF	147	49	196
TI	35	43	78
WS	309	91	400
TOTAL	1,669	1,143	2,812

Table 15: Number of each type of domain failure

Domain Failures				
Program	Non-Affine	Gotos	Subroutine Calls	Conditional Writes
AP	37	72	35	32
CS	114	60	47	4
LG	86	40	40	22
LW	21	12	7	1
MT	13	18	7	3
NA	93	38	11	8
OC	71	12	12	8
SD	64	26	23	9
SM	235	51	22	31
SR	12	10	2	4
TF	33	19	1	7
TI	37	20	5	0
WS	43	45	33	2
TOTAL	859	423	245	131

This is not because array references are always affine, it is because the lack of a true dependence can sometimes be proven without using the non-affine terms. Memory disambiguators can gracefully deal with non-affine information by conservatively ignoring a loop bound or an array dimension. Data-flow dependence analyzers cannot always make conservative approximations. Our current algorithm requires that all terms be affine. In the future, we believe that this can be extended. If, for example, we have a loop-constant, non-affine symbolic expression for a loop bound, we can treat the expression as a new symbolic variable. Also, if the non-affine terms are the same for each iteration of a loop corresponding to an unused index, we will still write the same locations in each iteration of the loop, and we should be able to conservatively ignore the non-affine terms.

Our second largest class of failures is due to goto statements inside loops. Our methods cannot handle arbitrary control flow as easily as summary based methods. We believe that we can extend our approach to other limited forms of control flow. For example, restructurers can sometimes eliminate goto statments. In the general case, we believe that combining our approach with the summary approaches should give the ideal solution.

Our domain can also be extended interprocedurally by utilizing summary information. This can lead to a loss of accuracy in the rare cases when read and write operations are interleaved across subroutine boundaries. We discuss interprocedural analysis in greater detail in the next chapter.

Finally, IF statements whose conditions are affine functions of the loop indices and symbolic constants can be easily incorporated into our framework. The IF statement would merely impose additional constraints on our memory disambiguator. For more general IF statements, we would have the same problem, and solution, as with arbitrary gotos.

4.9 Chapter Conclusions

Standard data dependence abstractions techniques are not sufficient for the more advanced optimizations being considered today. In particular, scalar optimizers have long used both data-flow and alias information to analyze programs. Parallelizers traditionally use only

data dependence analysis or, more precisely, affine memory disambiguation of array locations. A read statement is assumed to be dependent on a write if the read and write statements access overlapping locations. Data-flow analysis on individual array elements allows for more powerful optimizations.

In this chapter, we propose a new framework for data-flow dependence analysis using the abstraction of data-flow dependence vectors. We show how to derive data-flow dependence vectors from last write trees (LWTs), a general representation capturing the complete results from array data-flow analysis. Most array references are simple. Our empirical results suggest that most arrays either exhibit no reuse, or the array reuse occurs within those loops whose indices are unused by the array write reference function. This observation leads to an algorithm that can efficiently generate a last write tree and data-flow dependence vectors for these common cases.

As an example, we demonstrate the application of array data-flow analysis in array privatization. Using data-flow vectors and LWTs, we can easily integrate array privatization into parallelization. Replacing data-dependence vectors with data-flow vectors, the spurious dependences due to the reuse of the array locations are eliminated.

This chapter lays the foundation for new compiler optimizations that use sophisticated data-flow analysis on array elements. In addition to array privatization, our analysis techniques can also be used for other loop transformations based on dependence vectors, such as unimodular loop transformations and tiling or blocking. The need for data-flow analysis techniques is even more obvious in the compilation for message-passing machines. The high communication cost on these machines makes it absolutely necessary to minimize communication. Critical optimizations include reducing total communication volume by eliminating redundant data movements and reducing start-up overhead per message by blocking the communication. Such optimizations again need data-flow information on individual array elements.

Chapter 5

Interprocedural Analysis

In the previous chapters, we have described techniques for computing the dependences between pairs of references. To successfully parallelize programs, we need to compute the dependences between all array statements. To compute distance and direction vectors, we need to compare every array reference with every other reference to the same array, a process that could require $O(n^2)$ dependence tests. Computing data-flow dependence vectors is even worst. In the worst case, the cost of computing all the data-flow dependence vectors might grow exponentially with the number of array statements.

We showed in Chapter 2 that the cost of calculating distance and direction vectors between all array reference pairs in a subroutine is quite reasonable. To calculate all the dependence relations interprocedurally, we would have to fully inline the program. As we will show, this is completely infeasible.

An alternative approach is to summarize array accesses. We will describe several summary approaches. These summary algorithms approximate the array access behavior. Using approximate summary information, if we fail to parallelize a loop, it is not possible to know if the loop is inherently serial or if our summary was forced to make a conservative approximation. In addition, even a perfectly accurate summary is not sufficient for more advanced optimizations. While an accurate summary is sufficient for determining whether a loop can be run in parallel without modification, more information is required for array privatization and for loop transformations.

In this chapter, we introduce an approach that combines summary information with

subroutine inlining. We summarize whenever we can guarantee that our summary exactly represents the area of the array being accessed. If we cannot guarantee the exactness of our summary, we inline. Thus with the goal of parallelizing loops that contain subroutine calls, we are able to obtain the accuracy of inlining without the cost of inlining every subroutine call. In addition, because we are able to use inlining in cases of failure, we are able to use a very simple summary representation that can easily be calculated. In the end of the chapter, we discuss how to extend our approach to the more advance optimizations of array privatization and loop transformations.

5.1 Subroutine Inlining

With subroutine inlining, we replace every subroutine call with the body of the called subroutine. Using full inlining eliminates the interprocedural optimization problem by replacing the program with one very large subroutine.

In Fortran, inlining can result in some loss of information. For example:

```
program prog
do i
  call fred(i)
end do
end

subroutine fred(i)
dimension loc(100)
  ...
return
end
```

Subroutine `fred` accesses the local array `loc`. Each invocation of `fred` accesses a new local array and therefore there is no communication between the different iterations of the i loop in the main program. If we are not careful, after inlining, we may lose the

information that `loc` was local to `fred`, and we might unnecessarily sequentialize the loop.

Fortran allows for the reshaping of arrays across subroutine calls. Each subroutine accesses its arrays according to the locally declared shape. For example:

```
subroutine sub(a,n)
dimension a(10*n)
  call fred(a,n)
end do
end

subroutine fred(a,n)
dimension a(n,10)
do i = 1 to 10
  do j = 1 to n
    a(j,i) = ...
  end do
end do
return
end
```

After inlining we must transform the two dimensional array reference to `a` in subroutine `fred` into the equivalent one dimensional reference in subroutine sub. In this case, the affine reference `a(`j,i`)` in `fred` gets transformed into the non-affine `a(`$ni+j$`)`.

Some careful modifications to the inlining problem can reduce the amount of information lost. It is possible, for example, to note after inlining that the array `loc` is local to the scope of the loop body. Some of the other problems with inlining also exist with summarizers. Summarizers can also not deal with arbitrary array reshaping. We therefore feel that the main difficulties with inlining are its costs and not its accuracy. For the remainder of this chapter, we will consider inlining to be effectively completely accurate.

5.1.1 Full Inlining

In terms of accuracy, we consider full inlining to be the ideal solution. Full inlining, though, greatly increases the size of the program. In addition, the effects of this growth can be much larger than just the growth of the total program size. Most compiler analysis techniques work on each subroutine individually. Full inlining results in a program with just one subroutine. Even if the total program does not grow, the size of the average subroutine might grow tremendously. Table 16 gives several measures for the effects of full inlining on the PERFECT Club programs. In the first column, we show the ratio between the size of the inlined program and the size of the initial program. While for some programs, there is not much growth, for others the growth is huge. CSS and SMS, for example, grow by about a factor of 100 each. In the second column, we show the growth in the average subroutine size. While subroutine size growth is an intuitively appealing measurement, we feel that the ratios in the next two columns are more meaningful. Most analysis techniques cost $O(n^2)$ in the size of the subroutines in the worst case. In practice, most techniques cost $O(n \log n)$. For the third column, we sum up $n \log n$ over all the subroutines, where n is the size of each subroutine, and we show the ratio between this sum in the inlined and this sum in the original program. In the fourth column, we sum up the square of the size of the subroutines. This is an estimate for the worst case effect of inlining. We believe that the $n \log n$ column gives the most reasonable figure for the effect of inlining on efficiency. It is clear from this that full inlining is not realistic. Note that because of memory and time limitations, the experiment did not actually fully inline these programs, but given the size of each subroutine and the call graph, it is straightforward to calculate the effects of inlining.

Our experiment may slightly exaggerate the effects of inlining. Hall argues that scalar optimizers reduce the size of inlined object code by eliminating dead code and creating more specialized code [22]. She compares how much optimizers reduce object code size for inlined programs versus uninlined program. In her best examples, the optimizer is able to shrink inlined code by about 10% more than it was able to shrink uninlined code. While this is impressive, it is certainly not enough when programs grow by a factor of 100. In addition, the optimizer that shrinks the code must itself operate on the unreduced inlined code.

Table 16: Growth in PERFECT Club programs after full inlining

Growth After Full Inlining				
Program	Total Program	Average Subroutine	$n \log n$	n^2
AP	11.5	112.5	22.5	3,136.1
CS	94.8	2,445.1	197.7	292,946.0
LG	11.1	200.5	20.2	1,576.3
LW	1.6	27.4	2.3	21.4
MT	3.8	51.8	6.4	260.8
NA	1.6	23.2	2.4	37.3
OC	6.5	117.4	10.4	215.3
SD	2.1	19.1	3.6	86.9
SM	125.6	3,641.5	231.4	99,049.6
SR	1.4	18.1	2.1	30.9
TF	2.1	72.0	3.2	53.2
TI	1.0	2.0	1.1	2.0
WS	19.8	160.6	38.4	8,324.8

5.1.2 Restricted Inlining

It is clear that one cannot fullly inline a program. Thus all practical inliners must be restrictive and not inline every call. Hall utilizes two approaches [22]. For certain optimizations she restricts inlining based on the size of the code; large subroutines are not inlined. For other optimizations, she uses goal directed inlining; she attempts to inline sites that are likely to aid further optimization. Both these approaches might be very reasonable. Hall is able to, for example, eliminate most call statements within loops. Nonetheless, these approaches are very difficult to evaluate. It is not possible to know how much more parallelism would have been discovered if an uninlined site had instead been inlined.

An alternative approach is to not inline sites where we can prove that inlining is not beneficial. For example, we are only interested in parallelizing loops. If a call statement is not directly inside a loop nor in a subroutine that is called from inside a loop higher up in the call graph, then there is no point in inlining the call. In Table 17, we repeat the experiment of Table 16 but only inline calls inside *for* loops. A *for* loop is a SUIF

representation for non-while loops.

Table 17: Growth in PERFECT Club programs after inlining calls not inside for loops

Growth After Loop Restricted Inlining				
Program	Total Program	Average Subroutine	$n \log n$	n^2
AP	9.1	14.2	15.8	1,044.5
CS	4.0	4.4	5.2	60.2
LG	9.6	18.1	14.7	248.9
LW	1.6	2.5	1.9	6.0
MT	3.8	8.2	5.6	87.3
NA	1.1	1.3	1.2	1.4
OC	1.7	1.7	2.0	6.2
SD	1.2	1.9	1.4	2.5
SM	125.6	331.0	231.3	98,948.1
SR	1.0	1.2	1.1	2.3
TF	1.0	1.0	1.0	1.0
TI	1.0	1.3	1.1	1.7
WS	19.7	41.3	34.2	2,315.7

Using loops to restrict inlining reduces the expansion in subroutine size from inlining. Assume, for example, that a user is willing to allow inlining to increase compile time by a factor of about four or less. Using full inlining, eight of the thirteen programs have their $n \log n$ ratio increase by at least four, rounding to the nearest integer, most substantially more. With the loop restriction, six programs have such large $n \log n$ ratios.

Still, this is not good enough. No user is willing to have compile time increase by a factor of 200 for some programs, even if it barely increases for other programs. Memory utilization by the compiler is also a function of subroutine sizes. Even if the user is willing to wait, the compiler might run out of memory.

We had hoped to find additional effective pruning techniques that would have made restricted inlining sufficient by itself. Unfortunately, we were not able to find any that were sufficiently successful.

5.2 Summary Information

Summary information provides a method to reduce the number of dependence tests performed. We partition all the references. For each partition, we replace the set of references in the partition with a structure that summarizes the effects of all references in the partition. Rather than comparing every array reference with every other, a dependence analyzer need only compare every summary with every other.

We may use any criteria to partition the references. We might, for example, want to summarize all references within a single loop nest. Typically, though, we partition the references at the subroutine level. We summarize all the references within one subroutine. In this way, summarizing is an alternative to inlining. Rather than inlining a call, we annotate the call with the summary for the inlined subroutine.

There are many alternative summary representations. At one extreme, inlining can be regarded as a perfectly accurate summary that summarizes a subroutine with the code for the subroutine. At another extreme, we may use a scalar summary that denotes which variables are read and written in a subroutine. For example:

```
program prog
dimension a(100)
do i
  call fred(a,i)
end do
end

subroutine fred(a,i)
dimension a(100)
  a[i] = a[i]+1
return
end
```

Our summary for `fred` might contain the information that array `a` is both written and read. This loop can be executed concurrently since in each iteration of the i loop,

subroutine `fred` accesses a different region of the array. Such a conservative summary, though, would require us to assume that the whole array is accessed in each iteration and parallelization would be impossible.

Many summary representations, with varying degrees of accuracy have been proposed. Hall uses a variation of regular section descriptors, RSDs [22][11][23]. In this variation of RSDs, each dimension of each array is summarized separately. Each dimension is represented by either a constant symbolic expression; a range consisting of a lower bound, upper bound and step size; or $\perp$, signifying that the entire dimension is accessed.

Balasundaram advocates simple sections [6]. A simple section is a convex polytope with simple boundaries. A simple boundary is a hyperplane of the form $x_i = c$ or $x_i \pm x_j = c$. Thus a simple boundary is either parallel to a coordinate axis or at a 45 degree angle to a pair of coordinate axes.

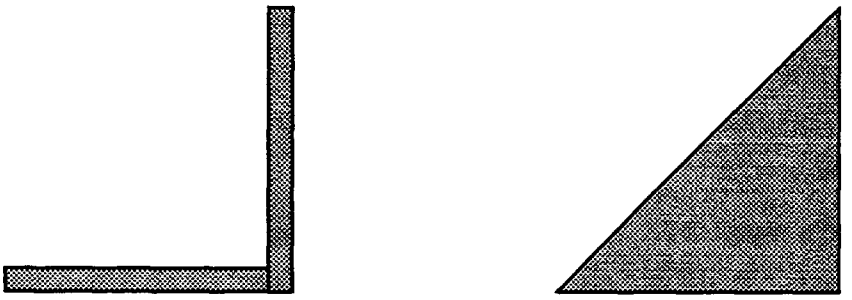
Triolet advocates a more accurate approach. He finds the convex hull of all the array references [51]. While this approach is more accurate, handling convex hulls can be considerably more expensive than the other two approaches.

All three of these approaches approximate the accessed regions. The regions described may be larger than the actual regions accessed. While these approaches are fairly accurate when describing single array accesses, even the convex hull approach may be inaccurate when describing the union of more than one fairly simple access pattern.

In Figure 15.a we show the region accessed by two array references; one of which accesses a row of the array and the other of which access a column. In Figure 15.b we show the resultant convex hull. For this example, the approximation could easily prevent parallelization.

As with approximate affine memory disambiguators, it is not possible to know when these algorithms approximate, and therefore it is very difficult to evaluate the effectiveness of these approaches in practice. If a loop is not parallelized, it is not possible to know if the loop is inherently serial or if the summarizer was too inaccurate.

As an alternative approach, Burke and Cytron keep a list of all the different array accesses [10]. They do not find the union of different accesses to the same array. While possibly as accurate as inlining, this approach could be just as inefficient. The number of array references should grow just as the object size grows with inlining.



a. two accesses b. resultant convex hull

Figure 15: Inaccuracy of convex hulls

5.3 Combining Summary Information and Inlining

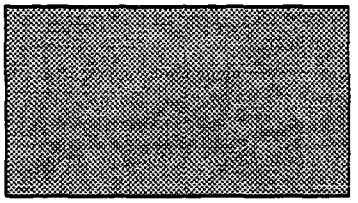
Our approach is to combine summary information with inlining. We use a simple summary, which we guarantee to exactly describe the regions accessed. If we are not able to exactly represent a region with our summary, we mark the summary as *bad*. Any subroutine call to a subroutine containing a bad summary will be inlined. We will show that our summary is applicable in enough of the common cases to prevent the large code growth of inlining. Thus our approach is as accurate as inlining while still retaining the efficiency of simple summarizers. By using inlining as a backup, we are able to use a very simple summary representation. It does not matter if our representation cannot accurately summarize a fairly common access pattern as long as our representation can accurately summarize most patterns.

Note that our exactness guarantee is flow insensitive. If, for example, subroutine `fred` is never called dynamically, we might incorrectly say that `fred` accesses a certain region while in fact the correct summary is that `fred` does not access any data. Most intraprocedural dependence analyzers are flow insensitive as well so we do not feel that this is unreasonable.

5.3.1 Multidimensional Rectangles

We chose as our summary representation, multidimensional rectangles. Each array variable has two such rectangles, one for the read accesses and one for the write accesses. Each dimension of the rectangle is represented with a minimum and maximum value. We only summarize when every element in the range is accessed and when the summaries for each dimension are decoupled. Thus, triangular or other non-rectangular shapes are not allowed. In Figure 16, we show a typical array reference with its associated summary.

```
do i = 1 to 10
  do j = 1 to 5
    a[i][j+1]
```



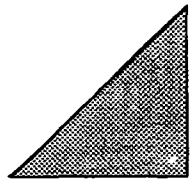
[1 to 10][2 to 6]

Figure 16: Example of multidimensional rectangle

In Figure 17, we show several examples of array references that cannot be summarized exactly by our algorithm. The presence of any such array reference forces us to inline. Figure 17.(a) shows a simple triangular region. Figure 17.(b) shows a diagonal region. Each dimension in the reference is separately summarizable, but the coupling of the two dimensions makes accurate summary impossible. In Figure 17.(c), we show a reference with a step size of two. Only half of the elements in the rectangle are accessed. Figure 17.(d) shows two rectangular references. Each reference can be accurately summarized, but the union of the two is not rectangular.

Multidimensional rectangles are similar to Hall's variation of RSDs. We've made a few simplifying restrictions. Hall allows general constant symbolic expressions. We restrict each side of our range to include at most one constant symbolic variable. Thus we could summarize the range [n to $2m$], but we could not summarize the range [1 to $n+m$]. This restriction both simplifies the union operation and cuts down on the amount of storage required to store the summary. Hall's ranges also contain a step size. Thus,

a) **do** i = 1 **to** 10
  do j = 1 **to** i
    a[i][j]



b) **do** i = 1 **to** 10
  a[i][i]

c) **do** i = 1 **to** 10
  a[2i]

d) **do** i = 1 **to** 10
  a[i][1], a[10][2]

Figure 17: Examples of non-rectangular regions

for example, she can represent the range [1 to n by 2]. We do not allow arbitrary steps in our representation. As a compromise, we allow conjugate pairs; pairs of ranges with a step size of two with each range offset from the other by one element. After the union operation, the step will be eliminated. For example:

```
do i = 1 to n by 2
  a(i)
  a(i + 1)
end do
```

Each reference accesses every other element, but the two references together access every element. We summarize the two references together by [1 to $n + 1$]. This can be easily implemented. If necessary, this can be easily generalized to any non-unit constant step size.

Our goal with multi-dimensional rectangles is not to have a completely accurate summary. Our goal is to make our interprocedural analysis as accurate as sophisticated intraprocedural affine analyzers. For example:

```
call fred
...
subroutine fred
do i = 1 to n^2
  a(i)
end do
```

The upper bound of the loop is not affine. As described, multidimensional rectangles cannot accurately describe this reference. We could inline the call to `fred`, but there is no benefit to inlining. Our intraprocedural affine memory disambiguator would ignore the non-affine term. Rather than marking the summary as bad, which would force inlining, we instead allow the value *unknown* in our range. Thus, we would say that this reference accesses locations [1 to *unknown*]. This captures the same information that is used by our intraprocedural analyzer.

The use of *unknown* ranges could easily be abused. We could mark every unsummarizable case as [*unknown* to *unknown*]; eliminating all inlining. Therefore, we are very careful to only use *unknown* ranges when using them does not result in any loss of information to our interprocedural analyzer. We only introduce them in three cases. First, we use *unknowns* if there is a non-affine term in either the array reference or the bound. Second, we use *unknowns* if either the reference or the bound refers to a non-constant symbolic variable. For example:

```
if (condition) then
  n = 3
else
  n = n + 3
end if
do i = 1 to n
  a(i)
end do
```

The variable n is not a constant variable. Our scalar optimizer was not able to express n as a function of other variables. Thus, we say the reference accesses locations [1 to *unknown*].

Finally, we allow *unknowns* when the step size is unknown. Normalizing such loops for dependence analysis will introduce non-affine terms.

In all other cases, we do not allow *unknown* ranges, and we mark any other types of unsummarizable ranges as bad. For example, non-rectangular regions, ranges with multiple symbolic constants and references with arbitrary constant step sizes are all marked as bad. In addition, *unknown* ranges are never introduced when calculating the union of two rectangles or when processing a subroutine call.

5.3.2 Union of Multidimensional Rectangles

For each subroutine, we must find the union of all the write and all the read references to the same array. The presence of symbolic variables complicates this process. For example:

```
do i = 1 to n
  a(i)
end do
a(n + 1)
```

Looking at the references individually, we get the two summaries [1 to n] and [$n+1$ to $n+1$]. It is fairly clear that the desired summary for the union is [1 to $n+1$], but what if $n < 1$. For example, say that n is 0. Since, $n < 1$, the first loop is never executed. The second reference is executed so the correct dynamic summary is [0 to 0]. The summary [1 to $n+1$] does not exactly describe the accessed region. Even worse, it is not conservative. The location [0] is accessed but is not covered by our summary.

There are two solutions to this problem. The first is to set the summary to bad in such cases. We do not feel that this is the right decision. All these cases are dynamically degenerate. A programmer does not often write loops where the upper bound is smaller than the lower bound. We should not sacrifice parallelism in the typical case because of problems with degenerate cases.

Instead, the approach we take when calculating unions is to assume that the upper bound of the loop is at least as big as the lower. Of course, we do not generate incorrect code, even for degenerate cases. Each time we make such an assumption, we are restricting the value of symbolic variables. We must remember these assumptions, propagate them upwards and check them at run time. We feel that this is reasonable. A practical parallelizer must already check variables dynamically to avoid parallelizing loops that are too small.

In addition to problems with degenerate cases, there are problems in deciding the order in which to calculate the union of multiple regions. When calculating the union of multiple regions, we require that each intermediate union be representable with multidimensional rectangles. It is possible that each individual region is representable, that the total union of all the regions is representable, but that the union of a subset of the regions is not representable. Take for example the four regions in Figure 18. Each of the four regions is a rectangle. The four regions taken together form a rectangle, but the union of any three regions does not form a rectangle. Assume that we first calculate the union of regions 1 and 2. To calculate the union of the four regions while keeping each intermediate region rectangular, we must next calculate the union of regions 3 and 4 and then calculate the union of the two intermediate unions. Finding the correct order to calculate unions is nontrivial. Sorting is not sufficient. Assume we sort the four regions in the order (1,2,3,4). After calculating the union of regions 1 and 2 together, how do we

know to next calculate the union of 3 and 4 rather than the union of 3 into the union of 1 and 2? In general, finding two elements out of n that have a valid union might require $O(n^2)$ attempts. Each successful union reduces by one the number of regions. Therefore, to calculate the union of all n elements might require $O(n^3)$ attempts. We felt that $O(n^3)$ might be too inefficient. Instead we developed some sorting heuristics that require fewer comparisons but might unnecessarily force us to set the union to bad.

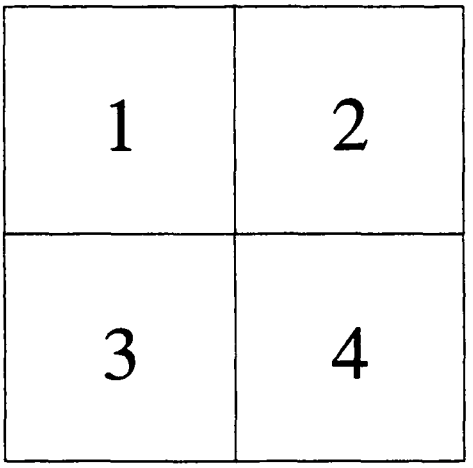


Figure 18: Union of several regions

5.3.3 Union of Subroutine Calls

When processing a subroutine call, we must calculate the union of the called subroutine with the summaries in our calling routine. Every summary implicitly depends on the context of its subroutine. Array a in the callee may no longer be array a in the caller. At each call, we must first convert the summary in the callee to match its new context and then we must calculate the unions of all the converted summaries in the callee with the summaries in the caller. This can be accomplished with the following four steps.

- Convert array names

- Convert symbolic variables
- Array reshaping
- Union of summaries

Converting Array Names

We must convert the name of each array, a, in the callee into the corresponding name in the caller. If array a is a global array, the name does not change. If a is a parameter, we must substitute the corresponding actual parameter for the formal. We do not summarize local arrays. Local arrays are reallocated at every subroutine invocation and therefore do not affect parallelization higher up the call graph. If the array a in the callee is converted into a local array in the caller, we throw out the corresponding summary.

Converting Symbolic Variables

The symbolic variables in the callee's summaries must also be converted when processing subroutine calls. For example, say that the summary for an array is [1 to n]. If n is a global variable, no conversion is necessary. If n is a parameter, we must substitute the corresponding actual parameter for the formal. All symbolic variables in our summaries must be subroutine constants. After converting n to, say, m, we must check if m is a subroutine constant. If it is, we are done. Otherwise, we check to see if m is an induction variable. If it is, we try to substitute its range for the summary. For example,

```
program prog
do m = 1 to 10
  call fred(a,m)
end do
end
```

```
subroutine fred(a,n)
a(n + 1)
return
end
```

Our summary for array a in subroutine fred is [$n+1$ to $n+1$]. When summarizing prog, we convert n to m, an induction variable that ranges from 1 to 10. Our summary for a in prog is therefore [2 to 11].

If the resultant region is non-rectangular or if m is neither a subroutine constant nor an induction variable, we set the summary of the calling subroutine to bad.

Array Reshaping

Fortran allows for arbitrary array reshaping across subroutines. Each array is in effect a pointer to a memory location. When accessing an array element, the array access is converted into a memory location using the dimension sizes of the array. For example, given the reference a(m,n) where a is declared with dimension a(a_{dim1},a_{dim2}), the location accessed is the base location of a + na_{dim1} + m (Fortran uses column major ordering). The various dimension sizes and even the number of dimensions may vary in different subroutines. For example:

```
program prog
dimension a(m,n,o)
...
call fred(a)
end
subroutine fred(a)
dimension a(n,m)
...
return
end
```

The main program accesses a as an $m \times n \times o$ array while the subroutine accesses a as an $n \times m$ array.

In addition, in Fortran one does not have to pass a pointer to the first element of the array; one can pass a pointer to any arbitrary location. For example, the following code passes an array starting at the fourth element of a to the subroutine.

```
program prog
dimension a(m)
...
call fred(a[4])
end

subroutine fred(a)
dimension a(m)
...
return
end
```

To summarize the main program, we must calculate the union of all its references to a with all of the subroutine's references to a. Before we can do this, we must reshape all of the subroutines summaries to take into account the new shape of a in the main program. Rectangular regions do not necessarily map into rectangular regions after arbitrary reshaping. For example:

```
program prog
dimension a(2,5)
...
call fred(a)
end
```

```
subroutine fred(a)
dimension a(10)
do i = 1 to 3
  a(i)
end do
return
end
```

Subroutine fred accesses the representable region [1 to 3]. In terms of the main program's shape, though, the accessed region is not rectangular; it is in fact one and a half columns. We cannot accurately summarize the main program. Even in cases where the mapped region would remain rectangular, it is not always easy to discover this.

We restrict the type of reshaping we can summarize to a few common styles that can be easily handled. For the types we allow, multidimensional rectangles always map into multidimensional rectangles, and the mapping between the two shapes can be easily computed. Any mapping that cannot be handled will force us to set the resultant summary to bad.

For simplicity, let us first assume that the number of dimensions in the calling and the called subroutine agree. Any reshaping can be described in general by the following code fragment:

```
program prog
dimension a(dp_1, dp_2, ..., dp_n)
...
call fred(a(o_1, o_2, ..., o_n))
end
subroutine fred(a)
dimension a(df_1, df_2, ..., df_n)
...
return
end
```

The last dimension of the array sizes, dp_n and df_n, do not affect the shape of the array. They are not used by the array reference functions. We therefore place no restrictions on their value. We require that all other dimension sizes match across the two routines, $dp_1 = df_1, dp_2 = df_2, \ldots, dp_{n-1} = df_{n-1}$. We also restrict the values of the offset, $o_1 \ldots o_n$. Setting o_n to x, for example, allows one to offset the array by x columns. The only restriction we place on o_n is that it be representable in our format. For example it cannot contain more than one symbolic constant. When any of the other offset variables is anything but one, a pointer is being passed to somewhere in the middle of a column. Therefore we require that $o_1 = 1, o_2 = 1, \ldots, o_{n-1} = 1$. With these restrictions, given a summary for subroutine `fred` of $[l_1$ to $h_1][l_2$ to $h_2]$...$[l_n$ to $h_n]$, to reshape the summary we merely need to add the offset into the last dimension to give us the resultant summary $[l_1$ to $h_1][l_2$ to $h_2]$...$[l_n + o_n - 1$ to $h_n + o_n - 1]$.

In general, we do not require the number of dimensions to agree. Given the following general code fragment:

program `prog`
dimension `a(`$dp_1, dp_2, \ldots, dp_n$`)`
...
call `fred(a(`$o_1, o_2, \ldots, o_n$`))`
end

subroutine `fred(a)`
dimension `a(`$df_1, df_2, \ldots, df_m$`)`
...
return
end

First, let us assume that the caller's array has more dimensions than the callee's, $n > m$. This style is frequently used by programmers to pass a column (or a multi-dimensional slice) into a subroutine. We restrict dimension sizes $1 \ldots m - 1$ as before,

$dp_1 = df_1, dp_2 = df_2, \ldots dp_{m-1} = df_{m-1}$. The dimension size m will no longer be the last dimension after reshaping. Thus, we must also require $df_m = dp_m$. The extra dimension sizes $dp_{m+1} \ldots dp_n$ are unrestricted. We restrict the first m offsets as before, $o_1 = 1, \ldots, o_{m-1} = 1$, and o_m is unrestricted. The extra offset dimensions $o_{m+1} \ldots o_n$ are used to select the slice. They are also unrestricted. With these restrictions, given a summary for subroutine `fred` of $[l_1$ to $h_1][l_2$ to $h_2]$...$[l_m$ to $h_m]$, our reshaped summary is $[l_1$ to $h_1]$...$[l_m + o_m - 1$ to $h_m + o_m - 1][o_{m+1}$ to $o_{m+1}]$...$[o_n$ to $o_n]$.

Now, let us assume that the callee has extra dimensions, $m > n$. The only case we handle is when the extra dimensions are of size one. Such a dimension is artificial. For example:

```
program prog
dimension a(d1)
...
call fred(a, 1)
end

subroutine fred(a, d2)
dimension a(d1, d2)
...
end do
return
end
```

Array `a` in routine `fred` is two dimensional, but when called from prog the second dimension size is one. A dimension whose size is one is an artificial array dimension. Whenever a callee has $m - n$ extra dimensions, we check to see if there are $m - n$ dimensions of size one. If there are, we eliminate these artificial dimensions from the summary and reshape as if the two dimension sizes were equal. While, this is applicable when any $m - n$ dimensions are equal to one, for simplicity we only check the first and last $m - n$.

Union of the Summaries

After doing the conversions and the reshaping, the context of all the summaries in the called subroutine is equivalent to the context in the calling subroutine. For our last step, we calculate the union of each summary region in the called subroutine with the corresponding summary in the calling subroutine.

5.4 Experimental Results

We have implemented our multidimensional summary algorithm and applied it to all the programs in the PERFECT Club. As our approach is as accurate as inlining, we judge its effectiveness by how much inlining is eliminated. In Table 18, we repeat the experiment of Table 17 but in addition to the loop pruning, we only inline when our summary is bad.

Table 18: Growth in PERFECT Club programs after inlining unsummarizable calls inside for loops

Growth After Summary and Inlining				
Program	Total Program	Average Subroutine	$n \log n$	n^2
AP	1.6	1.8	1.8	7.6
CS	3.2	3.3	3.9	33.0
LG	2.1	2.6	2.6	8.5
LW	1.2	1.3	1.3	2.3
MT	2.1	2.6	2.6	16.6
NA	1.0	1.0	1.0	1.0
OC	1.0	1.0	1.0	1.0
SD	1.1	1.4	1.2	1.6
SM	2.7	4.4	3.4	23.6
SR	1.0	1.2	1.1	2.3
TF	1.0	1.0	1.0	1.0
TI	1.0	1.1	1.1	1.7
WS	18.3	21.3	31.5	1,989.9

Our results have improved significantly. Only one program, WSS, has unacceptable growth. We looked at program WSS in detail. Two leaf subroutines that are called many

times, FFA99 and FFS99, linearize all their array accesses. In the comments, the array accesses are described in their multidimensional form. References that would be summarizable in their original form are not summarizable when linearized. To measure the effects of this linearization, we eliminated these subroutines and repeated our experiment. After eliminating these subroutines, the $n \log n$ entry for WSS only grows by a factor of 1.7. The square entry grows by a factor of 6.7. Thus if we were to implement the delinearization optimizations described in Chapter 3, we should be able to handle all the benchmarks in the PERFECT Club.

The cost of our approach must also take into account the time necessary to compute the summaries. In Table 19 we measured the amount of time it took, given a call graph, to summarize the programs. We compared this number to a standard interprocedural scalar optimizing compiler (f77 -O3). These number were taken on a DECstation 5000/200. The numbers for f77 -O3 are smaller than those in Chapter 2 because of the faster machine. The entries labeled * could not be measured due to bugs in version 1.31 of the MIPS Ultrix Fortran compiler. As can be seen, summarizing does not add significantly to the compile time.

Table 19: Total time of summarizing

Summarizing Time		
Program	Summarizing Time (in secs)	f77 -O3 (in secs)
AP	2.9	109.7
CS	61.9	*
LG	1.4	41.6
LW	0.8	19.6
MT	1.5	26.8
NA	2.6	79.5
OC	1.0	*
SD	0.9	37.0
SM	20.5	63.8
SR	2.5	75.4
TF	2.3	73.7
TI	0.1	6.0
WS	2.9	68.5

5.5 More Advanced Optimizations

Summary information allows us to detect parallelism in and compute direction vectors for all loops enclosing the summary. Summary information therefore enables us to parallelize outer loops that could not otherwise be parallelized. As described, summary information is not sufficient for calculating data-flow vectors. In addition, more sophisticated optimizations such as loop transformations require more than array summary information.

5.5.1 Interprocedural Data-flow Dependence Analysis

As we have described it, our approach is not sufficient to compute data-flow dependence vectors interprocedurally for two reasons; control flow analysis and the interleaving of reads and writes. Data-flow dependence analysis requires must information. An exact summary model such as ours is required, but in addition, we need to know that every write reference is in fact executed in every loop iteration. Control flow greatly complicates this analysis. Our simple data-flow dependence model in Chapter 4 does not handle control flow. We could easily extend this interprocedurally by marking each write summary that might be inexact due to control flow, but when our data-flow model is expanded to handle some control flow analysis, we may have to further enhance the summary information.

In addition, data-flow dependence analysis requires different information than simple dependence analysis. Take, for example, the following code fragment.

```
program prog
do i
  call fred1()
end do
do i
  call fred2()
end do
end
```

```
subroutine fred1()
do j
  a(j) =
  = a(j)
end do
return
end

subroutine fred2()
do j
  = a(j)
  a(j) =
end do
return
end
```

In the loop that calls `fred1`, there are no data-flow dependences carried by the i loop since every location read is first written. In the loop that calls `fred2`, the read statement comes lexically before the write, and the data-flow dependences are carried by the i loop. Both versions of `fred` generate the same summary information.

To compute data-flow dependences, we do not simply need to know which locations are read. Rather, we need to know which locations are read uninitialized. In addition, we still need to know which locations are written. When the write and read operations are not interleaved, it is easy to compute which locations are read uninitialized. Given a multidimensional rectangle for the locations read, we need to subtract the multidimensional rectangles for all the earlier writes. With multidimensional rectangles, subtractions is as easy as calculating unions. For example:

```
do i = 2 to 10
  a(i)= ...
end do
```

```
do i = 1 to 10
  ... = a(i)
end do
```

The first loop writes locations [2 to 10], the second loop reads locations [1 to 10]. Subtracting the first region from the second, we calculate that location [1 to 1] is read uninitialized. In general, for interleaved reads and writes, we must use the last write tree algorithm to find all the uninitialized reads.

As we mentioned in Chapter 4, we believe that in general our LWT algorithm from Chapter 4 must be combined with a data-flow summary algorithm. In regions of the program where our domain is applicable, the LWT algorithm is sufficient. In the presence of subroutine calls or control flow, we must integrate our LWT algorithm with summary based approaches.

5.5.2 Interprocedural Loop Transformations

Summary information is not sufficient for performing more advanced loop transformations useful in enhancing parallelism. For example:

```
program prog
do i
  call fred()
end do
end
```

```
subroutine fred()
do j
  do k
    a(j,k) = a(j,k)*c
  end do
end do
return
end
```

As written, the outer loop cannot be run in parallel, but the two inner loops can. For execution on a parallel machine, we would prefer to run the outer loops in parallel. It could be useful to interchange the loops:

```
program prog
do j
  call fred()
end do
end

subroutine fred()
do k
  do i
    a(j,k) = a(j,k)*c
  end do
end do
return
end
```

After summarizing `fred`, we have lost the information that loops j and k exist. We only know that in each iteration of the i loop, the array is being rewritten and reread,

generating a direction vector (*). Even summarizing the loop structure is not sufficient. For example:

```
program prog
do i
  call fred()
end do
end

subroutine fred()
do j
  do k
    a(j,k) = a(j,k)*c
  end do
end do
a(f1(i),1) = a(f2(i),1)*c
return
end
```

These loops cannot be interchanged because of the extra array statements in `fred` after the loops. Assuming that the original writes and reads accessed the entire array, both example access the same locations and have the same loop structures. No summary is sufficient. We need to know the exact relationship between the array accesses and the loops.

The amount and the nature of the information required is highly dependent on the optimization. In general, anything short of full inlining may be insufficient if we want to support every possible optimization. We see two possible approaches. Hall develops interprocedural loop transformations such as loop embedding that move loops across subroutine calls [22]. Once all the loops are moved, further loop transformations can be performed intraprocedurally. Alternatively, current loop transformation techniques tend to apply only in situations that are much more restrictive than simple parallelization.

Hence, it might be possible to inline all the locations where these transformations might be applicable without greatly increasing the resultant program size. Of course, this is highly dependent on the applicability of the loop transformations.

5.6 Chapter Conclusions

In this section, we have presented a new approach that combines inlining with summary information to accurately enable parallelization interprocedurally. Previous approaches have tended to rely only on either inlining or summary information. We have shown that full inlining is not practical since for some benchmarks, it vastly increases the program size. Restricted inlining either sacrifices potential parallelism or remains impractical. On the other hand, summary information sacrifices potential parallelism because of its inaccuracy.

By combining a simple summary scheme with inlining, we are able to retain the full accuracy of inlining for parallelization for a very reasonable cost. The exactness of our summary information allows us to consider using our approach for data-flow dependence analysis. We describe the extensions required when computing data-flow dependence information. Finally, we describe the complications that arise when performing interprocedural loop transformations and suggest two possible approaches.

Chapter 6

Conclusions

6.1 Contributions

This thesis has presented a new comprehensive approach to solving the data dependence problem. Specifically, we have made the following contributions. Much previous work has concentrated on solving the affine memory disambiguation problem. We have shown that it is possible to solve the affine memory disambiguation problem exactly and efficiently in practice. We have developed and implemented a suite of algorithms for this purpose. Solving the affine memory disambiguation problem has allowed us to develop a methodology for judging the effectiveness of the affine memory disambiguation domain restrictions. Any failure in our system is due to the limitations of the affine memory disambiguation model. By comparing our system to a dynamic, trace-based system that solves the data-flow dependence problem exactly, we have shown that data-flow dependence analysis is the key limitation to traditional data dependence analysis systems. Guided by these results, we have introduced a new approach for efficiently calculating data-flow dependence information. Finally, we have developed a new approach for interprocedural data-dependence analysis. By once again making our approximations explicit, we are able to develop an efficient algorithm that retains the accuracy of full inlining.

6.2 Future Work

6.2.1 Effectiveness of Parallelization

In Chapter 3, we studied the effectiveness of affine memory disambiguators in analyzing pairs of references. In reality, we are not interested in proving pairs independent, we are interested in parallelizing loops. Given a loop with 1,000 independent pairs, if our system was able to correctly prove 999 of the pairs to be independent, our experiment would say that our system was very successful. Nonetheless, we would have failed to exploit any of the available parallelism. In the future, we need to redo the experiment, using some measure of parallelism rather than the number of independent pairs. This is a much more difficult problem. It is clear what is meant by an independent pair. It is much less clear what is meant by a parallel loop. Is a loop with a reduction that can be partially parallelized considered a parallel loop? Parallelizability is heavily tied in with transformations. Measuring parallelizability does not depend only on analysis. It also depends on the suite of allowable transformations: reductions, privatization, loop distribution, etc. Much care will have to be taken in categorizing the failure cases to distinguish between failures in dependence analysis and weaknesses in the transformation system.

6.2.2 Data-Flow Analysis Domain

As we discussed in detail in Chapter 4, the domain of our data-flow dependence analysis algorithm is too narrow. We need to be able to extend our approach to handle non-affine terms, control flow and interprocedural analysis. With the extensions we described in Chapter 4, we believe that we can greatly increase the applicability of our approach.

6.3 Concluding Remarks

Data dependence analysis is the key step in parallelizing scientific codes in imperative languages. To parallelize loops, we must understand the array references inside loops.

In this thesis, we have used the data dependence problem to illustrate two key principles in compiler development. First, compiler optimizations can be driven by experimental results. For example, data dependence analysis is a difficult problem. Even the simpler problem of affine memory disambiguation is NP-Complete. Nonetheless, most problems seen in practice are much simpler than those conceived by theoretical limits. Using experimental results as a guide, we are able to tune our algorithms to be exact and efficient for the large majority of cases seen in practice. In addition, our experimental efforts have allowed us to focus on the key areas that limit parallelization. Theoretically, affine memory disambiguation has many limitations. Without performing the effectiveness study in Chapter 3, we might have concentrated our efforts on more thorough symbolic analysis rather than on the key problem of data-flow dependence analysis discussed in Chapter 4.

The second key principle is that it is important to make approximations explicit. Many algorithms have been proposed for the affine memory disambiguation problem. Different algorithms are advantageous for different types of inputs. By making our approximations explicit, by knowing when we approximate, we are able to effectively combine multiple algorithms. By making our approximations explicit, we are also usually able to guarantee that our algorithm returns exact results. The more advanced optimizations, such as array privatization, are not able to utilize conservative information. They need exact guarantees.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. Technical Report Rice COMP TR86-42, Department of Computer Science, Rice University, Nov 1986.

[4] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct 1987.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the ACM SIGPLAN PPOPP*, 1991.

[6] V. Balasundaram and K. Kennedy. A technique for summarizing data acess and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN PLDI*, pages 41–53, 1989.

[7] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[8] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[9] T. Brandes. The importance of direct dependences for automatic parallelism. In *Proceedings of International Conference on Supercomputing*, 1988.

[10] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162–175, 1986.

[11] D. Callahan and K. Kennedy. Analysis of interprocedureal side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, 1988.

[12] G. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[13] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Aug 1991.

[14] M. Berry et al. The PERFECT Club benchmarks: effective performance evaluation of supercomputers. Technical Report UIUCSRD Rep. No. 827, University of Illinois Urbana-Champaign, 1989.

[15] P. Feautrier. Array expansion. In *International Conference on Supercomputing*, pages 429–442, Jul 1988.

[16] P. Feautrier. Parametric integer programming. Technical Report 209, Laboratoire Methodologie and Architecture Des Systemes Informatiques, Jan 1988.

[17] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–52, Feb 1991.

[18] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN PLDI*, pages 15–29, 1991.

[19] E.D. Granston and A.V. Veidenbaum. Detecting redundant accesses to array data. In *Supercomputing '91*, pages 854–865, 1991.

[20] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software - Practice and Experience*, 20(2):133–155, 1990.

[21] M. R. Haghighat. Symbolic dependence analysis for high performance parallelizing compilers. In *3rd Workshop on Programming Languages and Compilers for Parallel Computing*, 1990.

[22] M. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, Apr 1991.

[23] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, Jul 1991.

[24] R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, Aug 1987.

[25] X. Kong, D. Klappholz, and K. Psarris. The i test: an improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, Jul 1991.

[26] D. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, 1978.

[27] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M.J. Wolfe. Dependence graphs and compiler optimizations. In *Eigth ACM Symposium on the Principles of Programming Languages*, Jan. 1981.

[28] James R. Larus. Estimating the potential parallelism in programs. In Aland David Padua, editor, *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, chapter 17, pages 331–349. MIT Press, 1991.

[29] H.W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[30] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the International Conference on Supercomputing (ACM), Washington, D.C.*, 1992.

[31] Z. Li and P. Yew. Practical methods for exact data dependency analysis. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.

[32] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan 1990.

[33] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in dependence testing phases of a vectorizer. In *Proceedings of Supercomputing 88*, pages 396–406, 1988.

[34] V. Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *Proceedings of the SIGPLAN PLDI*, pages 152–161, 1992.

[35] D.E. Maydan, S. P. Amarasinghe, and M.S. Lam. Data dependence and data-flow analysis of arrays. In *Conference Record of 5th Workshop on Languages and Compilers for Parallel Computing*, pages 283–292, 1992.

[36] D.E. Maydan, S. P. Amarasinghe, and M.S. Lam. Array data flow analysis and its application in array privatization. In *to appear in Proceedings of ACM POPL*, 1993.

[37] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN PLDI*, pages 1–14, 1991.

[38] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[39] V.R. Pratt. Two easy theories whose combination is hard. Technical report, Mass Institue of Technology, Sept 1977.

[40] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, 1991.

[41] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the SIGPLAN PLDI*, pages 140–151, 1992.

[42] H. Ribas. Obtaining dependence vectors for nested-loop computations. In *Proceedings of the International Conference on Parallel Processing*, 1990.

[43] C. Rosend. *Incremental Dependence Analysis.* PhD thesis, Rice University, March 1990.

[44] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[45] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependencies. In *Proceedings of International Conference on Parallel Processing*, pages II–145 to II–152, 1989.

[46] R. Shostak. Deciding linear inequalities by computing loop residues. *ACM Journal*, 28(4):769–779, Oct 1981.

[47] J.P. Singh and J.L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiporcessing: Tokyo, Japan*, 1991.

[48] R. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[49] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrated scalar optimization and parallelization. In *Proc. 4th Workshop on Programming Languages and Compilers for Parallel Computing*, Aug. 1991.

[50] Steven W.K. Tjiang and John L. Hennessy. Sharlit - a tool for building optimizers. In *Proceedings of the SIGPLAN PLDI*, pages 82–93, 1992.

[51] R. Triolet. Interprocedural analysis for program restructuring with parafrase. Technical Report CSRD Rep. No. 538, University of Illinois Urbana-Champaign, Dec. 1985.

[52] D. R. Wallace. Dependence of multi-dimensional array references. In *Proceedings of International Conference on Parallel Processing*, pages 418–428, 1988.

[53] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[54] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. Technical Report CS/E 90-015, Dept. of Computer Science and Engineering, Oregon Graduate Institute, August 1990.

[55] H. Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.